Executable Formal Specification of Programming Languages with Reusable Components

Submitted in fulfilment of the degree of Doctor of Philosophy by

L. Thomas van Binsbergen[1]

2018

Department of Computer Science

Royal Holloway, University of London

Supervisors	Examiners
Adrian Johnstone	Cliff B Jones
Elizabeth Scott	Matthew Hague

[1]Liewe Thomas van Binsbergen — `http://ltvanbinsbergen.nl` — ltvanbinsbergen@acm.org

Declaration of Authorship

I, Liewe Thomas van Binsbergen, hereby declare that this thesis and the work presented in this thesis is entirely my own, unless otherwise stated. Where I have consulted the work of others, this is clearly stated.

Signed:

Date:

17/09/2018

Acknowledgements

I would like to thank my supervisors Adrian Johnstone and Elizabeth Scott for the opportunities they have given me, their support, academic advice, and especially for challenging me to articulate my ideas carefully. I am grateful to Peter Mosses for inviting me to Swansea University on several occasions and giving me the opportunity to collaborate with the PLanCompS team. Of the PLanCompS team, I would like to thank Peter Mosses, Neil Sculthorpe and Casper Bach Poulsen for our collaborations and technical discussions.

The (social) life of a PhD student can be hard, and it is has been made much easier for me by Rob Walsh who welcomed me to the office and PhD student life, by the friendships with fellow PhD students in the department, and by the leisure of Royal Holloway's staff football. I want to especially thank my good friends Georgiana Lungu, Ionuţ Ţuţu, and Claudia Chiriţă for technical discussions and their support.

The collaborations with Duncan Mitchell and Johannes Kinder of Royal Holloway were very enjoyable and fruitful, for which I am grateful. I would also like to thank Martin Berger for our discussions on meta-programming and for inviting me to present my ideas at the University of Sussex.

This thesis marks the end of a long, privileged life as a student. I owe this privilege primarily to my parents Tonnie Kikkert and Janus van Binsbergen. With words I can only begin to express how grateful I am for their care and support, for believing in me, and giving me the freedom to develop myself according to my own plan.

Tonnie, Janus, en Jonas, met veel trots presenteer ik jullie dit werk. Bedankt voor de steun!

Abstract

Writing a formal definition of a programming language is cumbersome and maintaining the definition as the language evolves is tedious and error-prone. But programming languages have commonalities that can be captured once and for all and used in the formal definition of multiple languages, potentially easing the task of developing and maintaining definitions. Languages often share features, even across paradigms, such as scoping rules, function calls, and control operators. Moreover, some concrete syntaxes share patterns such as repetition with a separator, delimiters around blocks, and prefix and infix operators.

The PLanCompS project has established a formal and component-based approach to semantics intended to reduce development and maintenance costs by employing the software engineering practices of reuse and testing. This thesis contributes further, taking advantage of the advanced features of the Haskell programming language to define *executable* and *reusable* components for specifying both syntax and semantics.

The main theoretical contributions of this thesis are: a data structure for representing the possibly many derivations found by a generalised parser which significantly simplifies the specification of generalised parsing algorithms, a purely functional description of GLL parsing based on this data structure, a novel approach to combinator parsing that incorporates generalised parsing algorithms such as GLL in their implementation, and a novel and lightweight framework for developing modular rule-based semantic specifications (MRBS).

The main practical contributions of this thesis are: a combinator library for component-based descriptions of context-free grammars from which GLL parsers are generated, a compiler and interpreter for executing operational semantic specifications written in CBS (the meta-language developed by the PLanCompS project), an interpreter for an intermediate meta-language (IML) based on MRBS, and a translation from CBS specifications into IML that gives an operational semantics to CBS.

The practicality of the presented tools is evaluated through case studies.

Contents

Chapter 1

Introduction

1.1 Personal Reflection

The purpose of language is to convey meaning between speakers. The fact that sentences and words do not have a unique meaning is both a strength and a weakness of *natural language*. For example, the sentence "... and a man in a blue uniform gave me a ticket" has a different meaning when uttered after the sentence "I ran a red light in an attempt to be on time for my first date with Julie ..." than after the sentence "Julie and I went to the cinema for our first date ...".

Formal languages, like programming languages, are intended to be unambiguous in order to make precise, provable, and testable claims. In the case of programming languages, different programmers working on the same code should be confident about the code they are reading and modifying. Programmers are interested in the effect of running a certain piece of code on a particular machine, i.e. the behaviour of the program. Programmers have to be confident that the compiler they use generates machine instructions that exhibit the expected behaviour. Programming languages are given formal operational semantics in order to describe the behaviour of programs precisely. A formal operational semantics provides a precise, and machine-independent understanding of a language. We can talk about the exact behaviour of programs separately from the machine instructions that realise this behaviour. As such, a formal description of a programming language can act as a a contract between

compiler engineer and programmer.

Programmers can have tremendously successful careers without ever having considered formal descriptions of the programming languages they employ. One reason is that modern high-level programming languages are the produce of decades of language development. During this process, certain idioms have been established, e.g. looping constructs with breaks, function calls with return statements, and inheritance between objects. These idioms can be taught without formal underpinning. A second reason is experience. Just like when learning a natural language, programmers learn a programming language — or perhaps more accurately, a compiler for a language — by uttering sentences (running programs) and observing their effects.

Compiler engineers can be just as successful in their task of developing a compiler, without considering a formal semantics for the implemented language. In fact, only very few popular programming languages have a complete formal semantics. Moreover, the existence of a formal semantics does not imply that implementations are compliant to that semantics. In practice, experienced programmers and popular compilers are in agreement over the intended behaviour of most sensible programs. This common understanding is challenged by corner-cases, programs for which a language deviates from the established idioms or when two or more compilers have different interpretations.

Given these observations, why am I interested in formal semantics? In this thesis, I investigate an approach for describing semantics of languages in terms of reusable components. The components are given a formal semantics that can be understood in isolation. Crucially, the components can be *taught or understood without formal underpinning* and still *serve as the basis of a formal semantics*. A core assumption that underlies this thesis is that the process of language design benefits from such reusable components. Moreover, these components may prove useful in teaching programming languages at university level. If one understands certain language constructs generally, it is easy to learn specific languages with variations or specialisations of such constructs. Programmers with a high-level understanding of language constructs have the flexibility to switch between languages, and this kind of flexibility otherwise requires years of experience to be obtained.

1.2 Part 1: Syntax Analysis

Background Syntax analysis is one of the major success stories of theoretical computer science. There is a well-understood theory, namely that of regular and context-free grammars. The theory underpins the regular expression formalism and the BNF formalism for describing regular grammars and context-free grammars respectively. Efficient algorithms have been developed for these formalisms based on the automata associated with the grammar classes. The BNF formalism is at the heart of many parser generators like Yacc, Bison and Happy. Regular expressions are found in many programming languages like Perl, JavaScript and Python.

Parser combinators are a success story in their own right and are often taken as a prime example of the power of higher-order functional programming. A parser combinator expressions looks superficially like a BNF description of a grammar and behaves like a parser for that grammar. As a result, parser combinator expressions are easy to understand and maintain. Moreover, functions can be written that generate parser combinators, and parser combinators can be combined to form new ones. A parser combinator library typically provides a small set of elementary parsers and core combinators, together with combinators derived from them. Well-known parser combinator libraries are Parsec [Leijen and Meijer, 2001] and UU-Lib [Swierstra, 2009].

The success of parser combinators depends on the possibility to describe the syntax of a language (or data format) with a deterministic or unambiguous grammar. On the other hand, generalised parsing techniques like GLR [Tomita, 1985] and [Earley, 1970] enable writing parsers for the full class of context-free grammars, yielding all possible derivations of a sentence when a grammar is ambiguous. These general techniques have mostly been applied in the context of natural language processing, sometimes as part of a combinator library [Johnson, 1995, Frost et al., 2008].

Recent developments Generalised parsing is attractive to programming language researchers as it can simplify the formal definition of programming languages by reducing the gap between concrete and abstract syntax grammars. Since any context-free grammar admits a parser, the syntax of a language can be defined by a grammar that contains concrete information, like keywords and delimiters, whilst being sufficiently abstract for the purposes

of semantic analysis. Recent research has lead to improvements in derivation representation [Scott et al., 2007, Scott and Johnstone, 2010b], to principled approaches for ambiguity reduction [Van den Brand et al., 2003, Afroozeh et al., 2013], to generalised top-down parsing (GLL Parsing) [Scott and Johnstone, 2010a, Scott and Johnstone, 2013], and to parser combinator libraries that employ generalised parsing [Ridge, 2014, Izmaylova et al., 2016].

Contributions In part 1 of the thesis, we explore approaches to defining parser combinator libraries that employ generalised parsing. Chapter 2 gives a purely functional description of a GLL algorithm referred to as FUN-GLL. We explain GLL's data structures as abstract sets with basic operations, rather than specialised implementations, focussing on the algorithm's logic. We hope this alternative description makes GLL Parsing accessible to functional programming communities. The FUN-GLL algorithm and its description have been published as [Van Binsbergen et al., 2018]. The description benefits from a novel take on representing derivations, which is to use sets rather than trees, making it generally easier to turn recognisers into parsers. A detailed analysis of this alternative representation of derivation information is under review at the time of writing [Scott et al., 2019].

Taking into account the works of many authors, Chapter 3 discusses conventional parser combinators [Leijen and Meijer, 2001, Frost et al., 2008, Swierstra, 2009] and two methods found in literature [Johnson, 1995, Ridge, 2014] to generalising the conventional approach. In Chapter 4 we introduce our novel approach to combinators inspired by these methods. By letting combinator expressions represent BNF grammars explicitly, it is possible to extract grammar information easily, which is required for generalised parsing. The result is a collection of combinators for writing executable grammar specifications, referred to as BNF combinators. An explicit grammar object is computed from a combinator expression and given to the FUN-GLL algorithm. An implementation of these combinators is discussed in Chapter 5 and published as [Van Binsbergen et al., 2018].

1.3 Part 2: Interpretation

Background Several frameworks have been developed for defining the semantics of programming languages formally. In the framework of denotational semantics, program frag-

ments are structurally translated into objects within some semantic domain whose semantics is already understood [Mosses, 1990]. In an axiomatic semantics, a collection of axioms for a language is identified so that program properties can be proven by deduction [Hoare, 1969]. We focus on operational semantics, in which programs transition over time, potentially changing the state of an abstract machine at each step. Examples of frameworks for operational semantics are Plotkin's SOS [Plotkin, 2004b, Plotkin, 2004a], the $\mathbb{K}$ framework [Roşu and Şerbănuţă, 2014], and reduction semantics with evaluation contexts [Felleisen et al., 2009, Danvy and Nielsen, 2004]. Especially relevant to this thesis are MSOS and I-MSOS, modular variants of SOS for writing extensible language definitions, developed by [Mosses, 2004] and [Mosses and New, 2009]. Chapter 6 discusses the background material relevant to this thesis: Plotkin's SOS framework [Plotkin, 2004b] and Mosses modular variants MSOS [Mosses, 2004] and I-MSOS [Mosses and New, 2009].

Recent developments Despite the availability of semantic frameworks, it is not common practice that programming languages are developed together with a formal semantics. This may be due to the initial effort of writing a formal semantics and subsequently maintaining the formal semantics as the language evolves. As explained in "A History of Haskell" [Hudak et al., 2007], the requirement to maintain a formal definition discourages proposals for changes to the language.

As an attempt to reduce these efforts, the PLANCOMPS project[1] proposes a component-based approach to writing formal language definitions. The central idea behind the approach is that many programming languages, even languages from different paradigms, have a lot in common, and that these commonalities can be captured in a library of highly reusable fundamental language constructs, called funcons. The CBS language has been developed by PLANCOMPS for defining the syntax of programming languages in a variant of BNF, and the semantics of languages via a translation to funcons. The funcons themselves are formally defined in CBS via I-MSOS inference rules. The intention is that CBS language definitions are executable so that prototype implementations can be generated throughout the development cycle.

[1] http://plancomps.org

Contributions The main goal of part 2 of this thesis is to develop sound interpreters for funcon terms based on the formal definitions of funcons in CBS. The motivation is to support the intention of the PLANCOMPS project to generate prototype implementations from CBS language definitions. In this thesis we focus on operational semantic specifications of deterministic programming languages.

The main practical result is the Haskell Funcon Framework and its funcon term interpreters[2], generated from CBS specifications. The funcon term interpreters are used to verify CBS specifications, replacing the Prolog interpreters previously used by the PLANCOMPS project [Bach Poulsen and Mosses, 2014b]. The funcon term interpreters have been tested through unit-tests developed for individual funcons. Chapter 9 presents some of the implementation details of the framework, which have been extracted from papers about tool support for CBS [Van Binsbergen et al., 2016, Van Binsbergen et al., 2019].

The main theoretical result is the Modular Rule-Based Semantics (MRBS) framework for describing the operational semantics of programming languages. Compared to SOS and (I-)MSOS, MRBS is less generic and more specialised, with specifications that involve auxiliary semantic entities for modelling computational effects. MRBS has a notion of derivations formed by production rules that do not have to mention all auxiliary semantic entities, thereby improving the modularity of specifications, and interpreters are defined as algorithms that find such derivations. The IML language is a formalism for developing MRBS specifications. The strength of IML is that inferences rules are restricted to a form that admits a relatively straightforward mechanical treatment. MRBS and IML are introduced together in Chapter 7.

In Chapter 8, we show that rules written at a higher-level of abstraction can be translated into IML rules, taking CBS funcon and datatype definitions as an example. In this sense, IML is an Intermediate Meta-Language, and it is our expectation that inference rules written in other SOS based semantic frameworks, such as DynSem [Vergu et al., 2015], can be translated into IML as well. IML has been carefully designed to allow inference rules in different styles (e.g. small-step or big-step) and to be free of unnecessary restrictions.

The translation from CBS funcon definitions to IML can be seen to give a formal operational semantics to funcon definitions. We have used the translation to experiment with

[2]Plural, to include the language-specific extensions of the main interpreter.

implementation strategies and to build confidence in the definitions of funcons. We also discuss the implementation of novel funcons for Homogeneous Generative Meta-Programming (HGMP), expressing their behaviour in IML, and extending the Haskell Funcon Framework with manual implementations of these funcons. The funcons for HGMP and a case study that employs them are published as [Van Binsbergen, 2018].

1.4 Part 3: Evaluation

Part 1 and 2 of the thesis describe several formalisms supporting the development of deterministic programming languages with executable and reusable components for both syntax and semantics. Part 1 describes the core BNF combinators from which other reusable components can be derived. Part 2 discusses CBS funcon definitions and approaches to generating implementations of funcons. The BNF combinators, IML, and the CBS to IML translation have been implemented in Haskell tools. The tools are available as supplementary (see Section 1.5), together with the Haskell Funcon Framework, as `cabal` packages. In part 3 of these thesis, we evaluate the BNF combinators (`gll` package) and the funcon term interpreters (`funcons-tools` package). The runtime efficiency of these tools is evaluated in Sections 13.1 and 13.2 respectively.

We demonstrate, through case studies, that the reusable components provided by `gll` and `funcons-tools` can be used together to develop programming languages in literate Haskell form. The case studies are:

- Mini, a basic procedural language with exceptions and arrays (Chapter 11)
- Caml Light, a functional language with algebraic datatypes, pattern matching, mutable data, and exceptions (Chapter 12)

In Section 13.3 we reflect on the case studies and on using the BNF combinators and funcon terms interpreters generally.

1.5 Supplementary Material

The following tools and source files are available as supplementary material at `http://www.ltvanbinsbergen.nl/thesis`:

File	Description
`gll.tar.gz`	The implementation of GLL parsing with flexible BNF combinators
`iml-tools.tar.gz`	The IML interpreter and tools
`funcons-tools.tar.gz`	The Haskell Funcon Framework. Tools for executing funcon terms based on modular micro-interpreters
`funcons-values.tar.gz`	The implementation of the built-in types, values and value operations of CBS
`funcons-intgen.tar.gz`	The CBS to Haskell compiler, generating micro-interpreters from funcon definitions. The compiler also implements the CBS to IML translation
`CBS-beta-website.zip`	An archive containing the CBS source files of the funcons and the example languages as available at the time of writing. References in the thesis to "the CBS example languages" refer to the CBS specifications in the `Languages-beta` subfolder. References to the "current funcons" and "funcons in beta-release", refer to the funcons defined in the `Funcons-beta` subfolder
`CBS2IML.zip`	An archive containing the IML implementation of the current funcons, without static refocussing. The archive contains all the files necessary to execute tests and includes a large number of tests
`CBS2IML.iml`	concatenation of the files in `CBS2IML`. Simply give this file to the IML interpreter to execute a large number of tests
`CBS2IML-refocussed.zip`	Identical to `CBS2IML.zip`, except that funcon definitions have been translated with static refocussing
`CBS2IML.iml`	concatenation of the files in `CBS2IML-refocussed`. Simply give this file to the IML interpreter to execute a large number of tests
`funcons-lambda-cbv-mp.tar.gz`	An executable specification of a basic call-by-value lambda-calculus with meta-programming [Van Binsbergen, 2018]
`mini-reuse.tar.gz`	The Mini case study, including literate Haskell and test programs
`caml-light-reuse.tar.gz`	The Caml Light case study, including literate Haskell and test programs

Part I

Syntax Analysis

Chapter 2

Generalised Parsing

In the context of syntax analysis, a language is defined as the set of sentences that can be generated by applying the rules of the language's grammar. A derivation proves that a particular sentence is an element of a language, if the derivation steps are in accordance with the rules of the grammar. A parser is an algorithm that finds a derivation of an input sentence for a particular grammar, if one exists, and usually structures its output as a tree. A complete parser finds all possible derivations, and often complete parsers present these derivations in the form a forest, with sharing for efficiency. Generalised parsing involves the construction of (complete) parsers for arbitrary context free grammars.

In this thesis we consider complete parsing algorithms that take an arbitrary grammar, as well as a sentence, as input and yield all possible derivations of the sentence as output. We treat such algorithms as interpreters for the language of all grammars. This treatment underpins the discussion on parsing combinators in Chapters 3 and 4, in which combinator libraries are seen as defining embedded domain-specific implementations of the BNF grammar formalism.

This chapter formalises the aforementioned concepts and introduces a novel, unstructured data structure, called the *extended packed node set*, as an an alternative output for complete parsing algorithms. The extended packed node set is the output of choice for the parsing algorithms presented in this thesis. We give a purely functional generalised parsing algorithm, called FUN-GLL, an instance of generalised LL (GLL) parsing. The algorithm

operates on abstract data structures, modelled as mathematical sets, rather than concrete implementations. We hope this alternative description makes GLL parsing more accessible to functional programming communities.

2.1 Preliminaries

2.1.1 Symbols and Grammars

Given a finite set of *symbols* S, a *production* is a pair $\langle X, \alpha \rangle \in S \times S^*$ denoted as $X ::= \alpha$ (with X and α referred to as the production's left-hand side and right-hand side respectively). For a set of productions p, $N(p) = \{X \mid \langle X, \alpha \rangle \in p\}$ is the set of nonterminal symbols of p and $T(p) = \{s \mid \langle X, \alpha s \beta \rangle \in p\} \setminus N(p)$ is the set of terminal symbols of p. A *context-free grammar* is a pair $\gamma = \langle Z, p \rangle$, with $p \subset S \times S^*$ a finite set of productions and $Z \in N(p)$ the start symbol of the grammar[1]. We write *prods*(γ) for p. Hereafter we use 'grammar' to mean context-free grammar. The set $N(\langle Z, p \rangle)$ of nonterminals of a grammar is defined as $N(p)$ and the set $T(\langle Z, p \rangle)$ of terminals of a grammar is defined as $T(p)$. We omit the argument to N and T when it is clear from context to which grammar or set of productions is referred.

In this chapter, we let a sequence of n symbols be denoted by $s_1 s_2 \ldots s_n$, let the empty sequence be denoted by ϵ and sequence concatenation by juxtaposition. We use (possibly subscripted) X, Y and Z as placeholders for nonterminals; capital alphabetical characters for actual nonterminals; t as a placeholder for terminals; lowercase alphabetical characters for actual terminals; s for symbols, α, β, and δ for (possibly empty) sequences of symbols; p for sets of productions, γ for grammars; and u, v, and I for sequences of terminal symbols (also called sentences). If $I = t_0 \ldots t_{m-1}$ is a sentence, then I_k is the kth terminal in the sentence (t_k) and $I_{k,r}$ is the subsentence ranging from k to r (exclusive) $(t_k \ldots t_{r-1})$.

Figure 2.1 shows the example grammar used in this chapter. The example is taken from [Aycock and Horspool, 2002].

[1] This definition excludes grammars with repeated rules and with production-less nonterminals.

$$\gamma_{\text{AH}} = \langle S, \{S ::= AA, A ::= a, A ::= E, E ::= \epsilon\}\rangle$$
$$N(\gamma_{\text{AH}}) = \{S, A, E\}$$
$$T(\gamma_{\text{AH}}) = \{a\}$$

Figure 2.1: Running example grammar γ_{AH}.

2.1.2 Languages and Derivations

In the context of syntax analysis, a programming language is defined as the set of sentences that can be generated by applying the production rules of a what is often called a concrete grammar for the language. A concrete grammar describes the syntactically valid programs of the language and captures the structure of programs as they appear. When analysing semantics, a language may be defined as the set of trees that can be formed according to the productions of a possibly different grammar, often called an abstract grammar for the language. An abstract grammar leaves out concrete details, such as the delimiters around blocks and the texts of keywords, and captures the structure of programs in a way that is useful to describe their meaning.

In this chapter we define the set of all derivation trees of a grammar and define the sentences of a language as being those sequences of terminals found at the frontiers of derivation trees. Derivation trees capture the hierarchical structure embedded in sentences, as determined by a grammar, and are used to represent programs unambiguously. Trees are defined in the style of [Mosses, 1990].

In the context of generalised parsing algorithms, it is beneficial to use *binarised* parse trees, as binarised trees admit more efficient sharing [Scott et al., 2007]. For reasons that become clear later, we add indices to the labels of binarised parse trees. An intuitive explanation of the following definition is given afterwards.

Definition 2.1.1. Given a grammar $\gamma = \langle Z, p\rangle$, a *binarised parse tree* (BPT) is a pair $\langle\langle s_1 \dots s_m, l, r\rangle, t_1 \dots t_n\rangle$ with $s_1 \dots s_m \in S^*$, $l \in \mathbb{N}$, $r \in \mathbb{N}$, $l \leqslant r$, $m \geqslant 0$, $0 \leqslant n \leqslant 2$ and $t_1 \dots t_n$ binarised parse trees, such that:

$$
\begin{aligned}
m = 1, s_1 = Z \quad &\vee \quad \exists (X ::= s_1 \dots s_m \beta) \in p \\
n = 0, r = l &\iff m = 0 \\
n = 0, r = l + 1 &\iff m = 1, s_1 \in T(\gamma) \\
n = 1, t_1 = \langle \langle \beta, l, r \rangle, c_1 \rangle &\iff m = 1, s_1 \in N(\gamma), s_1 ::= \beta \in p \\
n = 2, t_1 = \langle \langle s_1 \dots s_{m-1}, l, k \rangle, c_1 \rangle, t_2 = \langle \langle s_m, k, r \rangle, c_2 \rangle &\iff m \geqslant 2
\end{aligned}
$$

The set $bpts(\gamma)$ is the smallest set containing all BPTs with respect to γ. For any BPT $t_0 = \langle \langle \alpha, l, r \rangle, t_1 \dots t_n \rangle$ we let $lb(t_0) = \langle \alpha, l, r \rangle$, $lbs(t_0) = \alpha$, $lbl(t_0) = l$, and $lbr(t_0) = r$ (l and r are referred to as the left and right extent of the tree respectively).

The first property of a BPT determines that each node is labelled with the right-hand side of a production, or that it is labelled with the start symbol. The second property determines that a node labelled with ϵ has equal indices ($l = r$) and has no children. The third property determines that a node labelled with a single terminal symbol has no children and that the difference between its indices is 1. The fourth property determines that a node labelled with a single nonterminal symbol has one child which is labelled with the same indices and with the right-hand side of one of the productions of the nonterminal. Any other node is labelled with a sequence of symbols of at least length 2 and has exactly two children whose labels 'compose' to form the label of their parent. These kinds of nodes show how a sequence of symbol derives a subsentence of the input sentence by inductively showing how its symbols derive parts of the subsentence, one symbol at a time (right child).

The *frontier* of a BPT t is the sequence of terminal symbols encountered when performing an order-respecting depth-first traversal on t.

Definition 2.1.2. For any BPT $t_0 \in bpts(\gamma)$, $frontier_\gamma(t_0) \in T(\gamma)^*$ is defined by:

$$
frontier_\gamma(\langle \langle \alpha, l, r \rangle, t_1 \dots t_n \rangle) = \begin{cases} \alpha & \textbf{if } n = 0 \\ frontier_\gamma(t_1) \dots frontier_\gamma(t_n) & \text{otherwise} \end{cases}
$$

Definition 2.1.3. A *derivation tree* of sentence $I \in T(\gamma)^*$, derived from $\alpha \in S^*$, is a BPT $t_1 \in bpts(\gamma)$ with $lbs(t_1) = \alpha$ and $frontier_\gamma(t_1) = I$ for some grammar γ. (It follows from the definition of $bpts(\gamma)$ and $frontier_\gamma$ that $lbr(t_1) = lbl(t_1) + |I|$.)

Figure 2.2 shows two valid BPTs with frontier a, respecting γ_{AH}. The language defined by a grammar is the set of all frontiers obtained from the set of all BPTs with a root symbol node labelled by the start symbol of the grammar and a left extent of 0.

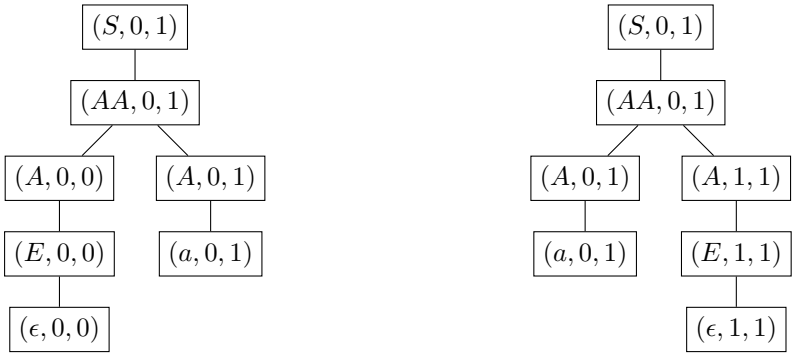


Figure 2.2: Binarised parse trees for the running example and sentence $I = a$.

Definition 2.1.4. The *language* of grammar $\gamma = \langle Z, p\rangle$ is the set of sentences $L(\gamma) = \{frontier_\gamma(t_1) \mid t_1 \in bpts(\gamma), lbs(t_1) = Z, lbl(t_1) = 0\}$

Definition 2.1.5. A nonterminal $X \in N(p)$ is *nullable*, denoted by $nullable_p(X)$, if it derives the empty sentence, i.e. $\epsilon \in L(\langle X, p\rangle)$

An algorithm is a *recogniser* for a grammar if it decides whether a given terminal sequence is a sentence of the language generated by the grammar.

A BPT not only tells us that a certain sentence can be derived, it tells us *how* this sentence can be derived from the productions of some grammar (reflected in the structure of the tree). This information is used when interpreting sentences according to the semantics of the language (Part 2 of this thesis). An algorithm is a *parser* for a grammar if it returns a BPT for a given input sentence, if one exists.

Big-step proof derivations Derivations are traditionally defined as sequences of derivation steps. The derivation steps can be formalised as a Plotkin style small-step transition system by giving logical inference rules (see [Plotkin, 2004b] and Chapter 6). The big-step style inference system in Figure 2.3 defines a relation (for every grammar γ), capturing transitions of sequence of symbols α into sequence of terminal symbols I in a single step $\langle \alpha, l, r\rangle \Rightarrow_\gamma I$. This system is insightful as it concisely and simultaneously captures the definition of BPTs and their frontiers. If there is a proof for $\langle \alpha, l, r\rangle \Rightarrow_\gamma I$ in the inference system, then there exists a BPT with label $\langle \alpha, l, r\rangle$ and frontier I. Conversely, if there is a BPT with label $\langle \alpha, l, r\rangle$ and frontier I, then there is a proof of $\langle \alpha, l, r\rangle \Rightarrow_\gamma I$. The proof of

$$\frac{t \in T(\gamma)}{\langle t, l, l+1\rangle \Rightarrow_\gamma t} \quad \text{(TERM)}$$

$$\frac{(X ::= \alpha) \in \mathit{prods}(\gamma) \qquad \langle \alpha, l, r\rangle \Rightarrow_\gamma u}{\langle X, l, r\rangle \Rightarrow_\gamma u} \quad \text{(NTERM)}$$

$$\langle \epsilon, l, l\rangle \Rightarrow_\gamma \epsilon \quad \text{(EPS)}$$

$$\frac{\langle s_1 \ldots s_{n-1}, l, k\rangle \Rightarrow_\gamma u \qquad \langle s_n, k, r\rangle \Rightarrow_\gamma v \qquad n > 1}{\langle s_1 \ldots s_{n-1} s_n, l, r\rangle \Rightarrow_\gamma uv} \quad \text{(SEQ)}$$

Figure 2.3: Rules defining $\Rightarrow_\gamma$ for the grammar γ.

this equivalence is given in Appendix A.1. In our reasoning we often rely on the inference system, assuming that it is clear (from intuition, or from the proof in the appendix) how this reasoning relates to the BPTs that underpin the formal definitions of languages.

Ambiguity A sentence can be derived in more than one way. Rule NTERM may be applied with multiple choices of α to give the same conclusion. Similarly, SEQ might be applicable for more than one choice of k and give the same conclusion. This shows that different BPTs may have the same frontier and root label, i.e. there may be different derivations of the same sentence. A grammar γ is *ambiguous* if there is a sentence $I \in L(\gamma)$ such that there is more than one BPT with frontier I. The example grammar is ambiguous, as shown by the two alternative BPTs given in Figure 2.2 for sentence $I = a$. This particular ambiguity arises from the two possible choices $k = 0$ or $k = 1$ in the application of SEQ to prove $AA \Rightarrow_{\gamma_{\text{AH}}} a$. We define the set $\mathit{derivs}(\gamma, I)$, the set that contains all derivation trees of I.

Definition 2.1.6. Given a grammar $\gamma = \langle Z, p\rangle$, the set of derivations of I from $\alpha \in S$ is $\mathit{derivs}_\gamma(\alpha, I) = \{t_1 \mid t_1 \in \mathit{bpts}(\gamma), \mathit{lb}(t_1) = \langle \alpha, 0, |I|\rangle, \mathit{frontier}_\gamma(t_1) = I\}$. We write $\mathit{derivs}_\gamma(I)$ as a shorthand for $\mathit{derivs}_\gamma(Z, I)$.

An algorithm is a *complete parser* for a grammar if given an input sentence it returns all derivations of the sentence according to the grammar. A *general and complete parsing procedure* is a procedure that given an arbitrary grammar and an input sentence returns all possible derivations of that sentence according to the grammar.

$$
\begin{aligned}
e_1 &= \langle A ::= a\cdot, 0, 0, 1\rangle & e_6 &= \langle S ::= AA\cdot, 0, 1, 1\rangle \\
e_2 &= \langle S ::= A \cdot A, 0, 0, 1\rangle & e_7 &= \langle A ::= AA\cdot, 0, 0, 1\rangle \\
e_3 &= \langle A ::= E\cdot, 0, 0, 0\rangle & e_8 &= \langle S ::= AA\cdot, 0, 0, 0\rangle \\
e_4 &= \langle S ::= A \cdot A, 0, 0, 0\rangle & e_9 &= \langle E ::= \cdot, 0, 0, 0\rangle \\
e_5 &= \langle A ::= E\cdot, 1, 1, 1\rangle & e_{10} &= \langle E ::= \cdot, 1, 1, 1\rangle
\end{aligned}
$$

Figure 2.4: A set of EPNs for the running example grammar γ_{AH} and sentence $I = a$.

$$
\begin{aligned}
\gamma_{\text{INF}} &= \langle S, \{S ::= ES, S ::= a, E ::= \epsilon, E ::= e\}\rangle \\
N(\gamma_{\text{INF}}) &= \{S, E\} \\
T(\gamma_{\text{INF}}) &= \{a, e\}
\end{aligned}
\qquad
\begin{aligned}
&\langle S ::= ES\cdot, 0, 0, 1\rangle \\
&\langle S ::= E \cdot S, 0, 0, 0\rangle \\
&\langle E ::= \cdot, 0, 0, 0\rangle \\
&\langle S ::= a\cdot, 0, 0, 1\rangle
\end{aligned}
$$

Figure 2.5: A set of EPNs embedding infinitely many derivations of the sentence $I = a$.

There are combinations of grammars and input sentences for which exponentially many (with respect to the length of the input sentence), or even infinitely many, derivations exist. It is therefore impossible for a complete parsing procedure to enumerate all possible derivations, for any combination of grammar and input sentence, in worst-case polynomial time. In practice, complete parsers produce a parse forest with sharing to efficiently represent all derivation trees. This forest often contains some subtrees that are not part of any of the derivation trees for the input sentence. This is not a problem, as parse forests are easily filtered to preserve only those derivation trees that represent full derivations of the input sentence (see also Appendix A.4). In fact, some complete parsers employ disambiguation strategies whilst filtering parse forests with the intention of preserving just one derivation tree that (hopefully) corresponds to the interpretation of the programmer. The topic of disambiguation is not studied in this thesis, but it will recur.

2.1.3 Extended Packed Nodes

Nodes in BPTs are labelled with a prefix α of the right-hand side of some production[2] $X ::= \alpha\beta$. We could have chosen to label nodes with productions instead, given a way of

[2]With the exception of the root node, although a root node labelled with X can be viewed as being labelled with the right-hand side of an auxiliary production $S ::= X$ for some fresh S.

'highlighting' α (in order to distinguish labels of parent and child). A *grammar slot* is a triple $\langle X, \alpha, \beta \rangle \in S \times S^* \times S^*$, denoted as $X ::= \alpha \cdot \beta$, that can be used for this purpose. ($X ::= \cdot$ is written when $\alpha = \beta = \epsilon$.) Rather than using grammar slots as labels for nodes, we use grammar slots in what we call *extended packed nodes*. An extended packed node is a quadruple $\langle g, l, k, r \rangle$ with g a grammar slot and $0 \leqslant l \leqslant k \leqslant r$ natural numbers called the *left extent*, *pivot* and *right extent* respectively.

Definition 2.1.7. Given a grammar γ, an *extended packed node* (EPN) is a quadruple $\langle X ::= \alpha \cdot \beta, l, k, r \rangle$, with $X ::= \alpha\beta \in \mathit{prods}(\gamma)$, $l \leqslant k \leqslant r$ natural numbers (the left extent, pivot, and right extent respectively).

The indices l, k, and r indicate that α is matched by the substring $I_{l,r}$, of some input sentence I, and that the last symbol s of α is matched by $I_{k,r}$ (or $l = k$ if $\alpha = \epsilon$).

A set of EPNs can act as an efficient representation of one or more BPTs. The intuition is that the information in the EPN reflects the choices that lead to the conclusion that a particular sentence I is in the language of some grammar. There are two kinds of choices: the choice of a production $X ::= \alpha$ when applying rule NTERM in the big-step inference system, and the choice of k when applying SEQ. The former is reflected in the grammar slot of an EPN. The latter is reflected in the natural numbers l, k, and r, that determine that u is the subsentence of I ranging from l to k ($I_{l,k}$), and v is the subsentence of I ranging from k to r ($I_{k,r}$) (k is called pivot because it splits sentences).

Definition 2.1.8. Given a set of extended packed nodes Δ, the set $\mathcal{T}(\Delta)$ is the smallest set of BPTs such that:

$$\langle X ::= \cdot, l, k, r \rangle \in \Delta \implies \langle \langle \epsilon, k, r \rangle, [] \rangle \in \mathcal{T}(\Delta)$$
$$\langle X ::= \alpha t \cdot \beta, l, k, r \rangle \in \Delta, t \in T(\gamma) \implies \langle \langle t, k, r \rangle, [] \rangle \in \mathcal{T}(\Delta)$$
$$X ::= \alpha \in \mathit{prods}(\gamma), t_1 = \langle \langle \alpha, l, r \rangle, c_1 \rangle \in \mathcal{T}(\Delta) \implies \langle \langle X, l, r \rangle, t_1 \rangle \in \mathcal{T}(\Delta)$$
$$\langle X ::= \alpha s \cdot \beta, l, k, r \rangle \in \Delta, \alpha \neq \epsilon, t_1 = \langle \langle \alpha, l, k \rangle, c_1 \rangle \in \mathcal{T}(\Delta), t_2 = \langle \langle s, k, r \rangle, c_2 \rangle \in \mathcal{T}(\Delta)$$
$$\implies \langle \langle \alpha s, l, r \rangle, t_1 t_2 \rangle \in \mathcal{T}(\Delta)$$

For $\mathcal{T}(\Delta)$ to be well-defined we must have that $r = k$ for all $\langle X ::= \epsilon, l, k, r \rangle \in \Delta$ and $r = k + 1$ for all $\langle X ::= \alpha t \cdot \beta, l, k, r \rangle \in \Delta$. Figure A.2 shows a set of EPNs embedding the trees of Figure 2.2. Figure 2.5 shows a set of EPNs embedding infinitely many binarised parse trees.

In the subsequent sections we discuss generalised and complete parsing procedures that compute sets of extended packed nodes embedding all possible derivation trees.

2.1.4 Discussion

We have given a big-step inference systems in which derivations correspond to binarised parse trees, therefore deviating from a more conventional definition of derivations based on derivation steps.

A Shared Packed Parse Forests (SPPF) is a graph that embeds a forest of parse trees efficiently through sharing. Scott, Johnstone and Economopoulos have shown that a binarised representation enables more sharing in SPPFs so that Binarised SPPFs (BSPPFs) have worst-case cubic space complexity [Scott et al., 2007]. For further details on (B)SPPFs we refer to [Scott and Johnstone, 2010b, Scott and Johnstone, 2013]. The notion of extended packed node is derived from the packed nodes of BSPPFs[3].

We introduce the extended packed node as a level of indirection that significantly simplifies the discussion of the parsing procedures in this chapter, as well as the parser combinators of Chapter 3 and 4. We can define post-processing algorithms that given a set of extended packed nodes produce a particular outcome, e.g. a parse tree, a parse forest, an SPPF, a semantic value, or a set of semantic values, and use these post-processors with any of our parsing procedures. In Appendix A.4 we show how a BSPPF can be constructed from a set of extended packed nodes. The constructed BSPPF contains no spurious derivations; it contains no derivations of subsentences of I that are not used in any of the derivations of I. In Chapter 4 we discuss a post-processor based on the semantic actions[4] of a combinator expression. The algorithm in Chapter 4 is derived from Ridge who uses a data structure similar to extended packed nodes (called "Earley productions") for the same purpose [Ridge, 2014].

In the next section we begin our description of parsing algorithms building up to formulations based on descriptors (similar to "Earley items"). As we shall see, algorithms based on descriptors are easily extended to compute sets of extended packed nodes.

[3] An extended packed node is simply a packed node with its parent's left and right extent.

[4] Parser generators often enable arbitrary code fragments, referred to as semantic actions, to be inserted into production rules. These code fragments are added verbatim to the generated parser and typically perform compilation or interpretation on the fly. Parser combinator libraries often enable the insertion of semantic actions into combinator expressions for the same purposes.

2.2 Recursive Descent Parsing

Recursive Descent Parsing (RD) is a technique for (manually or mechanically) writing top-down parsers based on the description of a context-free grammar. RD covers a collection of algorithms rather than one particular algorithm.

RD parsers (RDPs) have in common that: every symbol is implemented by a piece of code; these pieces of code can be placed in sequence, implementing a production's right-hand side; choice between (the pieces of code that implement) a nonterminal's productions is implemented by branching control-flow. The details of an RDP depend on the chosen algorithm and the language in which the parser is written (the host language). A nonterminal is typically implemented by a procedure (that we call a *parse function*) which, after one of its branches has been fully executed, relies on the host language's call-stack to return to the next piece of code in the sequence from which the parse function was called.

In basic RD, parse functions choose a single production (branch) to execute. Different RD algorithms choose between alternative productions differently. For example, the choice can be made based on lookahead. Lookahead involves (pre-)computing for each alternative the set of terminals it is capable of matching initially, and checking, during a parse, whether the next terminal in the input sentence is a member of that set. Lookahead can thus be used to rule out alternatives efficiently. In general, lookahead cannot be used to rule out each, or all but one, alternative. The class of LL(k) grammars is defined to contain those grammars for which it holds that, with $k > 0$ terminal symbols lookahead, no two alternatives are simultaneously applicable. After choosing an alternative, different forms of backtracking can be used to revert the decision and choose another alternative instead.

2.2.1 Generalising Recursive Descent Parsing

Earley notes about his generalised parsing algorithm — referred to as "Earley's Algorithm" — that it can be seen as a top-down parser by which all possible parses are performed simultaneously in such a way that common execution paths appear only once [Earley, 1970]. By "a parse" Earley means a single execution of a top-down parser, resulting from a certain combination of choices between alternatives. A different combination of choices results in a different parse. We make the analogy between top-down parsing and algorithms similar to

Earley's algorithm more precise by considering a particular style of RDPs.

In this style, parse functions have an integer parameter and return an integer. Both integers are indices into the input sentence. We assume that each parse function has a local variable for remembering the argument given to the parse function when it was called. We can interpret a grammar slot $X ::= s_1 \ldots \cdot s_j \ldots s_m$ as a unique identifier for a position in the source code of F, the parse function implementing nonterminal X. The production $X ::= s_1 \ldots s_j \ldots s_m$ uniquely identifies the code of a branch in F and the dot identifies a position within that branch, namely the code for the jth symbol of the production. The triple $\langle X ::= s_1 \ldots \cdot s_j \ldots s_m, l, k\rangle$ represents the state of a machine executing F, if we treat l as the value for the local variable and k as the value of the parameter. Such a triple is a *descriptor*[5].

Definition 2.2.1. Given a grammar $\gamma = \langle Z, p\rangle$, a *descriptor* is a triple of the form $\langle X ::= \alpha \cdot \beta, l, k\rangle$, with $X ::= \alpha\beta \in p$, and $l \leqslant k$ natural numbers (referred to as the left extent and index respectively).

A parser performing "all possible parses", ignoring lookahead, is modelled by a set $\mathcal{U}(\gamma, I)$ of descriptors as follows. If Z is the start symbol of γ, then $\langle Z ::= \cdot\delta, 0, 0\rangle$ has to be in $\mathcal{U}(\gamma, I)$, for all productions $Z ::= \delta \in P(\gamma)$ (all productions of the start symbol as left-hand side are considered). If a parse reaches terminal t of production $X ::= \alpha t\beta$, indicated by some descriptor $\langle X ::= \alpha \cdot t\beta, l, k\rangle$ in $\mathcal{U}(\gamma, I)$, then the parse continues if and only if $I_k = t$. If a parse reaches nonterminal Y of production $X ::= \alpha Y\beta$, indicated by some descriptor $\langle X ::= \alpha \cdot Y\beta, l, k\rangle$ in $\mathcal{U}(\gamma, I)$, then all productions with left-hand side Y are to be considered. If a parse reaches the end of a production, indicated by a descriptor $\langle X ::= \delta\cdot, k, r\rangle$ in $\mathcal{U}(\gamma, I)$, then $I_{k,r}$ can be derived from X and the parse that resulted in this alternative of X being considered may recommence. In fact, all parses that have reached and will reach nonterminal X with index k may continue with index r. This idea is formalised in the following definition.

Definition 2.2.2. Given a grammar $\gamma = \langle Z, p\rangle$ and a sentence I, the set $\mathcal{U}(\gamma, I)$ is the smallest set of descriptors such that:

[5] A descriptor is effectively an Earley item with an added index. In Earley's algorithm, a descriptor $\langle g, l, k\rangle$ appears as an item $\langle g, l\rangle$ in the state set Σ_k.

$$Z ::= \delta \in p \implies \langle Z ::= \cdot\delta, 0, 0\rangle \in \mathcal{U}(\gamma, I) \tag{R(1)}$$
$$\langle X ::= \alpha \cdot t\beta, l, k\rangle \in \mathcal{U}(\gamma, I), t = I_k \implies \langle X ::= \alpha t \cdot \beta, l, k+1\rangle \in \mathcal{U}(\gamma, I) \tag{R(2)}$$
$$\langle X ::= \alpha \cdot Y\beta, l, k\rangle \in \mathcal{U}(\gamma, I), Y ::= \delta \in p \implies \langle Y ::= \cdot\delta, k, k\rangle \in \mathcal{U}(\gamma, I) \tag{R(3)}$$
$$\langle X ::= \alpha \cdot Y\beta, l, k\rangle \in \mathcal{U}(\gamma, I), \langle Y ::= \delta\cdot, k, r\rangle \in \mathcal{U}(\gamma, I)$$
$$\implies \langle X ::= \alpha Y \cdot \beta, l, r\rangle \in \mathcal{U}(\gamma, I) \tag{R(4)}$$

A set that satisfies $R(1) - R(4)$ is referred to as a closed set of descriptors. A minimal closed set of descriptors is necessarily equal to $\mathcal{U}(\gamma, I)$.

The following definition captures the connection between descriptors and EPNs. To use Earley's terminology: there is an EPN whenever the dot of a grammar slot is 'moved across a symbol' [Earley, 1970] between two descriptors.

Definition 2.2.3. Given a grammar γ and a set of descriptors $\mathcal{U}$, the set $\Delta(\mathcal{U})$ is the smallest set of extended packed nodes such that:

$$\langle X ::= \alpha t \cdot \beta, l, k+1\rangle \in \mathcal{U} \implies \langle X ::= \alpha t \cdot \beta, l, k, k+1\rangle \in \Delta(\mathcal{U}) \tag{P(1)}$$
$$\langle X ::= \alpha \cdot Y\beta, l, k\rangle \in \mathcal{U}, \langle Y ::= \delta\cdot, k, r\rangle \in \mathcal{U}$$
$$\implies \langle X ::= \alpha Y \cdot \beta, l, k, r\rangle \in \Delta(\mathcal{U}) \tag{P(2)}$$
$$\langle X ::= \cdot, l, r\rangle \in \mathcal{U} \implies \langle X ::= \cdot, l, l, r\rangle \in \Delta(\mathcal{U}) \tag{P(3)}$$

The EPNs with a slot of the form $X ::= \cdot$ are a convenience; they avoid the need for special cases for nullable nonterminals in certain definitions.

2.2.2 Discussion

In this chapter, we discuss *descriptor processors*: parsing procedures computing a set of descriptors by processing descriptors sequentially. Sections 2.3 and 2.4 discuss descriptor processors that compute exactly the set $\mathcal{U}(\gamma, I)$. This is a simplification, however, enabling us to focus on the principles behind the algorithms. In practice, generalised parsing algorithms like Earley's algorithm and GLL parsers do not process all the descriptors in $\mathcal{U}(\gamma, I)$, as, for example, they use lookahead to avoid descriptors that have no chance of success (are not part of a complete parse).

The presented parsing procedures compute the set $\Delta(\mathcal{U}(\gamma, I))$ alongside $\mathcal{U}(\gamma, I)$, as it is simple and efficient to do so. Appendix A.4 explains how a complete BSPPF without spurious derivation trees is constructed from $\Delta(\mathcal{U}(\gamma, I))$, showing that our parsing procedures

can be extended to yield all derivation trees.

2.3 Descriptor Processing

Based on the definition of $\mathcal{U}(\gamma, I)$, we design an algorithm, called Closing a Descriptor Set (CDS), in the form of a closure algorithm. We show that CDS computes $\mathcal{U}(\gamma, I)$ and $\Delta(\mathcal{U}(\gamma, I))$ and use CDS as a basis to explain GLL Parsing. CDS is essentially Earley's algorithm but considers descriptors in an unspecified order whereas Earley's algorithm processes descriptors in order of increasing index (a comparison between CDS and Earley's algorithm is made in §2.3.4).

2.3.1 Closing a Descriptor Set (CDS)

Procedure 1 (CDS-PROC)**.** The inputs are sentence $I \in T^*$ and grammar $\gamma = \langle Z, p \rangle$. The goal is to compute $\mathcal{U}(\gamma, I)$. This is achieved by repeatedly inspecting the set of currently processed descriptors $\mathcal{U}$ to 'discover' descriptors until there are no more 'new' descriptors. Initially, $\mathcal{U} = \{(Z ::= \cdot\beta, 0, 0) \mid Z ::= \beta \in \gamma\}$. Set $\mathcal{U}$ is closed by repeatedly executing one of the following actions, until $\mathcal{U}$ stabilises:

1. If $\langle X ::= \alpha \cdot t\beta, l, k \rangle \in \mathcal{U}$, for any terminal symbol t with $t = I_k$ add $\langle X ::= \alpha t \cdot \beta, l, k{+}1 \rangle$ to $\mathcal{U}$

2. If $\langle X ::= \alpha \cdot Y\beta, l, k \rangle \in \mathcal{U}$, for any nonterminal symbol Y, add $\langle Y ::= \cdot\delta, k, k \rangle$ to $\mathcal{U}$, for all $Y ::= \delta \in \gamma$

3. If $\langle Y ::= \delta \cdot, k, r \rangle \in \mathcal{U}$ and $\langle X ::= \alpha \cdot Y\beta, l, k \rangle \in \mathcal{U}$, add $\langle X ::= \alpha Y \cdot \beta, l, r \rangle$ to $\mathcal{U}$

CDS-PROC can be concretised by a worklist algorithm that runs on a worklist $\mathcal{R}$ containing the descriptors that require *processing* and that processes each descriptor only once. A descriptor is processed by inspecting its slot and determining which of the actions it enables (if any).

Action 1 is only executed when a descriptor of the form $\langle X ::= \alpha \cdot t\beta, l, k \rangle$ (t terminal) is processed. If $t = I_k$, then the descriptor $\langle X ::= \alpha t \cdot \beta, l, k + 1 \rangle$ is added to $\mathcal{R}$. We call this the **match** action.

Action 2 is only executed when a descriptor of the form $\langle X ::= \alpha \cdot Y\beta, l, k\rangle$ (Y nonterminal) is processed. All the descriptors corresponding to all valid choices of δ are added to $\mathcal{R}$. We call this the **descend** action.

Action 3 is executed when a descriptor of the form $\langle Y ::= \delta\cdot, k, r\rangle$ or $\langle X ::= \alpha \cdot Y\beta, l, k\rangle$ (Y nonterminal) is processed. In the former case, all the descriptors $\langle X ::= \alpha Y \cdot \beta, l, r\rangle$, for valid choices of X, α, β, and l, are added to $\mathcal{R}$. We call this the **ascend** action. In the latter case, all the descriptors $\langle X ::= \alpha Y \cdot \beta, l, r\rangle$, for valid choices of r, are added to $\mathcal{R}$. We call this the **skip** action.

Processing a descriptor of the form $X ::= \alpha \cdot Y\beta$ requires a choice between the descend and skip actions. At most one of these actions is capable of increasing the size of $\mathcal{U}$. This is because a skip action on $X ::= \alpha\cdot Y\beta$ only adds additional descriptors if a descend action on that descriptor has already been performed and repeating a descend action does not result in more descriptors. The shape of a descriptor, and the contents of $\mathcal{U}$, therefore determine uniquely which of match, ascend, descend, or skip is to be executed in order to process that descriptor. And as we shall explain later, it is only necessary to process each descriptor once. These observations enable the following straightforward formalisation of CDS as a worklist algorithm.

2.3.2 Formalisation

Formalisation 1 (CDS-FORM)**.** Function *cds* receives a grammar $\gamma = \langle Z, p\rangle \in \Gamma$ and a sentence $I \in T^*$ as input, and computes $\mathcal{U}(\gamma, I)$ and $\Delta(\mathcal{U}(\gamma, I))$. The extended packed nodes are computed whilst descriptors are processed, rather than post-processing $\mathcal{U}(\gamma, I)$ to obtain $\Delta(\mathcal{U}(\gamma, I))$.

$$\begin{aligned} &\mathit{cds}(\gamma, I) = \mathit{loop}_{\gamma,I}(\{\langle Z ::= \cdot\beta, 0, 0\rangle \mid Z ::= \beta \in p\}, \emptyset, \emptyset) \\ &\quad \textbf{where } \langle Z, p\rangle = \gamma \end{aligned}$$

The core of the algorithm is formed by the recursive function $\mathit{loop}_{\gamma,I}$, whose first argument is a worklist $\mathcal{R}$. The second and third arguments of $\mathit{loop}(\gamma, I)$ are sets of descriptors and extended packed nodes. The descriptors are those that have already been processed ($\mathcal{U}$) and the extended packed nodes are those that have been 'discovered' so far ($\mathcal{E}$).

$$loop_{\gamma,I}(\mathcal{R},\mathcal{U},\mathcal{E}) = \begin{cases} \langle \mathcal{U},\mathcal{E}\rangle & \textbf{if } \mathcal{R}=\emptyset \\ loop_{\gamma,I}(\mathcal{R}\cup\mathcal{R}_1\setminus\mathcal{U}',\mathcal{U}',\mathcal{E}\cup\mathcal{E}_1) & \text{otherwise} \\ \quad \textbf{where } \langle \mathcal{R}_1,\mathcal{E}_1\rangle = process_{\gamma,I}(d,\mathcal{U}') & \\ \quad \textbf{and} \quad \mathcal{U}' = \{d\}\cup\mathcal{U} & \end{cases}$$
$$\textbf{where } d = select(\mathcal{R})$$

Function *loop* is defined such that $loop_{\gamma,I}(\mathcal{R},\mathcal{U},\mathcal{E}) = \langle \mathcal{U},\mathcal{E}\rangle$ if $\mathcal{R}=\emptyset$ and otherwise there is a $d\in\mathcal{R}$ and $loop_{\gamma,I}(\mathcal{R},\mathcal{U},\mathcal{E}) = loop_{\gamma,I}(\mathcal{R}\cup\mathcal{R}_1\cup\mathcal{U}',\mathcal{U}',\mathcal{E}\cup\mathcal{E}_1)$, where $\mathcal{R}_1$ and $\mathcal{E}_1$ are the results of processing d and $\mathcal{U}'$ is the set of processed descriptors extended with d. Function *process* executes a *match*, *descend*, *skip*, or *ascend* action leading to possible new descriptors $\mathcal{R}_1$ and extended packed nodes $\mathcal{E}_1$. Termination of *loop* is guaranteed by the fact that for every grammar and input sentence there are finitely many descriptors that can be discovered by *process*. Function *select* can be any function such that $select(\mathcal{R})\in\mathcal{R}$ if $\mathcal{R}\neq\emptyset$.

$$process_{\gamma,I}(\langle X ::= \alpha\cdot s\beta,l,k\rangle,\mathcal{U}) =$$
$$\begin{cases} match_{\gamma,I}(\langle X ::= \alpha\cdot s\beta,l,k\rangle) & \textbf{if } s\in T \\ descend_{\gamma,I}(s,k) & \textbf{if } R=\emptyset \\ skip_{\gamma,I}(\langle X ::= \alpha s\cdot\beta,l,k\rangle,R) & \text{otherwise} \end{cases}$$
$$\quad \textbf{where } R = \{r \mid (s ::= \delta\cdot,k,r\rangle\in\mathcal{U}\}$$
$$process_{\gamma,I}(\langle Y ::= \delta\cdot,k,r\rangle,\mathcal{U}) = \langle \mathcal{R}_1,\mathcal{E}_1\cup\mathcal{E}_2\rangle$$
$$\quad \textbf{where } \langle \mathcal{R}_1,\mathcal{E}_1\rangle = ascend(K,r)$$
$$\quad \textbf{and} \quad \mathcal{E}_2 = \{\langle Y ::= \cdot,k,r,r\rangle \mid \delta=\epsilon\}$$
$$\quad \textbf{and} \quad K = \{\langle X ::= \alpha Y\cdot\beta,l,k\rangle \mid \langle X ::= \alpha\cdot Y\beta,l,k\rangle\in\mathcal{U}\}$$

A set R is computed to determine whether to execute a *skip* or *descend* action, when processing a descriptor of the form $\langle X ::= \alpha\cdot s\beta,l,k\rangle$ with $s\in N$. The set R contains r if and only if there is a production $s ::= \delta$ whose symbols match the sentence $I_{k,r}$, indicated by a descriptor $\langle s ::= \delta\cdot,k,r\rangle\in\mathcal{U}$. If there are such r, a descend action on Y must have taken place; $\langle Y ::= \cdot\delta,k,k\rangle$ must have been processed in order to get $\langle Y ::= \delta\cdot,k,r\rangle\in\mathcal{U}$. If there are no such r a (possibly redundant) descend action will be executed.

#	**processing**	**action**	**new**	$\mathcal{E}$
1	$\langle S ::= \cdot AA, 0, 0)$	**descend**	$2, 4$	
2	$\langle A ::= \cdot a, 0, 0)$	**match**	3	e_1
3	$\langle A ::= a\cdot, 0, 1)$	**ascend**	6	e_2
4	$\langle A ::= \cdot E, 0, 0)$	**descend**	5	
5	$\langle E ::= \cdot, 0, 0)$	**ascend**	8	e_9, e_3
6	$\langle S ::= A \cdot A, 0, 1)$	**descend**	$7, 12$	
7	$\langle A ::= \cdot a, 1, 1)$	**match**	-	-
8	$\langle A ::= E\cdot, 0, 0)$	**ascend**	9	e_4
9	$\langle S ::= A \cdot A, 0, 0)$	**skip**	$10, 11$	e_7, e_8
10	$\langle S ::= AA\cdot, 0, 1)$	**ascend**	-	-
11	$\langle S ::= AA\cdot, 0, 0)$	**ascend**	-	-
12	$\langle A ::= \cdot E, 1, 1)$	**descend**	13	
13	$\langle E ::= \cdot, 1, 1)$	**ascend**	14	e_{10}, e_5
14	$\langle A ::= E\cdot, 1, 1)$	**ascend**	10!	e_6

Figure 2.6: Execution trace of CDS-FORM with **ascend** > **match** > **skip** > **descend**.

$$
\begin{aligned}
&\mathit{match}_{\gamma,I}(\langle X ::= \alpha \cdot t\beta, l, k\rangle) = \\
&\quad \begin{cases} \langle \{\langle X ::= \alpha t \cdot \beta, l, k+1\rangle\}, \{\langle X ::= \alpha t \cdot \beta, l, k, k+1\rangle\}\rangle & \textbf{if } t = I_k \\ \langle \emptyset, \emptyset\rangle & \text{otherwise} \end{cases} \\
&\mathit{descend}_{\gamma,I}(Y, k\rangle = \langle \{\langle Y ::= \cdot\delta, k, k\rangle \mid Y ::= \delta \in \gamma\}, \emptyset\rangle \\
&\mathit{ascend}_{\gamma,I}(K, r\rangle = \mathit{nmatch}(K, \{r\}) \\
&\mathit{skip}_{\gamma,I}(d, R) = \mathit{nmatch}(\{d\}, R) \\
&\mathit{nmatch}_{\gamma,I}(K, R) = \langle \mathcal{R}_1, \mathcal{E}_1\rangle \\
&\quad \textbf{where } \mathcal{R}_1 = \{\langle X ::= \alpha Y \cdot \beta, l, r\rangle \mid \langle X ::= \alpha Y \cdot \beta, l, k\rangle \in K, r \in R\} \\
&\quad \textbf{and} \quad \mathcal{E}_1 = \{\langle X ::= \alpha Y \cdot \beta, l, k, r\rangle \mid \langle X ::= \alpha Y \cdot \beta, l, k\rangle \in K, r \in R\}
\end{aligned}
$$

Function *nmatch* is given a set of triples K of the form $\langle X ::= \alpha Y \cdot \beta, l, k\rangle$, with the same non-terminal Y and integer k, and possibly different X, α, β, and l. For all $\langle X ::= \alpha Y \cdot \beta, l, k\rangle \in K$ and $r \in R$ new descriptors $\langle X ::= \alpha Y \cdot \beta, l, r\rangle$ are created. Function *nmatch* is called either with a singleton set K (*skip*) or a singleton set R (*ascend*). The definitions of *skip*, *ascend*, and *nmatch* are inspired by Johnson's memoisation combinator [Johnson, 1995], which is discussed in more detail in §3.3.2.

2.3.3 Completeness

By construction, *cds* computes a subset of $\mathcal{U}(\gamma, I)$ (soundness), for every grammar γ and sentence I, as all the descriptors in the result are justified by the rules of Definition 2.2.2. It

#	**processing**	**action**	**new**	$\mathcal{E}$
1	$\langle S ::= \cdot AA, 0, 0\rangle$	**descend**	$2, 3$	
2	$\langle A ::= \cdot E, 0, 0\rangle$	**descend**	4	
3	$\langle A ::= \cdot a, 0, 0\rangle$	**match**	8	e_1
4	$\langle E ::= \cdot, 0, 0\rangle$	**ascend**	5	e_9, e_3
5	$\langle A ::= E\cdot, 0, 0\rangle$	**ascend**	6	e_4
6	$\langle S ::= A \cdot A, 0, 0\rangle$	**skip**	7	e_8
7	$\langle S ::= AA\cdot, 0, 0\rangle$	**ascend**	-	-
8	$\langle A ::= a\cdot, 0, 1\rangle$	**ascend**	$9, 12$	e_2, e_7
9	$\langle S ::= A \cdot A, 0, 1\rangle$	**descend**	$10, 11$	
10	$\langle A ::= \cdot E, 1, 1\rangle$	**descend**	13	
11	$\langle A ::= \cdot a, 1, 1\rangle$	**match**	-	-
12	$\langle S ::= AA\cdot, 0, 1\rangle$	**ascend**	-	-
13	$\langle E ::= \cdot, 1, 1\rangle$	**ascend**	14	e_{10}, e_5
14	$\langle A ::= E\cdot, 1, 1\rangle$	**ascend**	12!	e_6

Figure 2.7: Execution trace of CDS-FORM with **skip** > **descend** > **match** > **ascend**.

is more complicated to argue that all descriptors in $\mathcal{U}(\gamma, I)$ are in the result (completeness). Every descriptor required by $R(1)$ (see Definition 2.2.2) is added as part of the initialisation. Every descriptor in the final set of descriptors has been processed and each processed descriptor results in either a match, descend, ascend, or skip action. The outcome of the match and descend actions do not depend on the contents of $\mathcal{U}$ at the time they are called, so they will add all the descriptors required by $R(2)$ and $R(3)$. The outcome of the skip and ascend actions do depend on the contents of $\mathcal{U}$. Thus, it seems, executing a skip or ascend action a second time to the same descriptor may lead to new descriptors, if $\mathcal{U}$ grew in between. However, if a skip action misses a descriptor at a given time, this descriptor is added later by a subsequent ascend action. The same holds vice versa, and thus all descriptors required by $R(4)$ are added. Appendix A.2 formalises and proves the claim that *csd* computes the set $\mathcal{U}$.

Appendix A.3 gives a proof that *cds* computes $\Delta(\mathcal{U}(\gamma, I))$, i.e. $\langle \mathcal{U}(\gamma, I), \Delta(\mathcal{U}(\gamma, i))\rangle = cds(\gamma, I)$.

Figures 2.6 and 2.7 show two alternative executions of CDS-FORM, both computing $\mathcal{U}(\Gamma_{\text{AH}}, I)$ and $\Delta(\mathcal{U}(\Gamma_{\text{AH}}, I))$ with $I = a$. The execution of Figure 2.6 discovers descriptor $\langle S ::= AA\cdot, 0, 1\rangle$ through a skip action, whilst the execution of Figure 2.7 does so through an ascend action.

2.3.4 Comparison with Earley's algorithm

The CDS algorithm is inspired by Earley's algorithm [Earley, 1970] and Pingali and Bilardi's description of Earley's algorithm [Pingali and Bilardi, 2015]. Earley's algorithm uses a set of "states" E_n for each possible index $0 \leqslant n \leqslant |I|$ into input sentence I, where each state can be represented by a pair of the form $\langle X ::= \alpha \cdot \beta, l \rangle$. CDS and Earley's algorithm are similar in the sense that when CDS processes a descriptor of the form $\langle X ::= \alpha \cdot \beta, l, k \rangle$, Earley's algorithm processes a state of the form $\langle X ::= \alpha \cdot \beta, l \rangle$ in the state set E_k. CDS generalises Earley's algorithm by processing descriptors in any order, rather than in any order *within* a state set and computing the state sets in ascending order. The most important difference between CDS-FORM and Earley's algorithm is that Earley's algorithm does not have an action analogous to skip, whereas Earley's **predictor** action is analogous to descend, **completer** to ascend, and **scanner** to match.

As we have seen in Section 2.3.3, the skip action is important for the completeness of CDS. As explained next, the skip action is redundant when processing descriptors in an order with ascending pivots and when the grammar has no nullable nonterminals. However, as Earley noted, a problem arises with the ascend (completer) action when a grammar has nullable nonterminals. When a descriptor of the form $\langle Y ::= \delta \cdot, k, r \rangle$ is processed, the algorithm searches for descriptors of the form $\langle X ::= \alpha \cdot Y\beta, l, k \rangle \in \mathcal{U}$ to create new descriptors with 'the dot moved over Y'. It follows that $k < r$ if there are no nullable nonterminals in the grammar (because δ is not nullable). If all previous descriptors were processed in an order with ascending pivots, then, following $k < r$, all descriptors of the form $\langle X ::= \alpha \cdot Y\beta, l, k \rangle$ must already be in $\mathcal{U}$. However, if Y is a nullable nonterminal it may be that $k = r$ (e.g. if $\delta = \epsilon$). In this case, there is no guarantee that all such descriptors have been added.

Earley solved this problem by suggesting an implementation that keeps track of ascend actions that may not have added all necessary descriptors so that subsequent descend actions may yield the missing descriptors [Earley, 1970]. Grune and Jacobs suggest to keep executing ascend and descend actions until no new descriptors are found [Grune, 2010]. This approach does not fit easily in our formalisation because it may involve processing the same descriptor more than once. Aycock and Horspool give the simplest solution [Aycock and Horspool, 2002]: when performing the descend action on $\langle X ::= \alpha \cdot Y\beta, l, k \rangle$, and when Y is nullable, add $\langle X ::= \alpha Y \cdot \beta, l, k \rangle$ to $\mathcal{R}$ (as well as the usual

descriptors arising from a descend action).

2.4 GLL Parsing

Like RDP, GLL Parsing is a technique for (manually or mechanically) writing, in this case, complete top-down parsers. Like RDPs, GLL parsers can be based on different algorithms, each with usage or performance benefits. GLL parsers can be written in any host language, although the specifics of how aspects of the underlying algorithm are implemented may vary considerably across host languages. The main complication is *continuation management*.

2.4.1 Continuation Management

A parse function of a complete RDP parser should execute all of its branches to be complete[6]. And, rather than a single integer (right extent), a parse function must be able to produce more than one right extent (1), because multiple of its branches may succeed with different right extents. As well as finding multiple right extents, a parse function must have multiple return locations or *continuations* (2), when it is called from different call sites. To realise this, a generalised RDP parser needs a more powerful form of continuation management[7] than is provided by standard call-stacks.

Scott and Johnstone's GLL parsers use a graph in which continuations are encoded in nodes and edges [Scott and Johnstone, 2010a]. In a functional programming setting, Johnson suggests a sophisticated form of memoisation which involves remembering the continuations of the parse functions as well as multiple return values [Johnson, 1995].

Scott and Johnstone explain GLL Parsing by means of a parser generator which writes parsers in a low-level pseudo-code containing labels and GOTO statements. This specific treatment of continuations makes it somewhat difficult to imagine GLL parsers in host languages that do not support such constructs. In this chapter we do not focus on (manually or mechanically written) GLL parsers. Instead, we give a parsing procedure that shows how sentences are parsed in accordance to an input grammar. The chosen strategy for finding all derivations is similar to that of a particular GLL parsing algorithm called

[6]The branches may be guarded by lookahead tests.

[7]We chose this term for referring, in a more general setting, to what is known as continuation passing in the functional programming community.

RGLL [Scott and Johnstone, 2016]. We formalise the algorithm using abstract mathematical objects to model continuations. Labels and GOTO statements give a specific concretisation of this model. In Chapter 3 on parser combinators, we see that parse functions themselves can concretise continuations in a setting with higher-order functions.

2.4.2 The Graph Structured Stack

We model a continuation as a pair $\langle s, l \rangle$ of a slot s and left extent l (essentially a descriptor missing its index). GLL parsers manage continuations themselves, rather than relying on a host language's call-stack. A call-stack can be represented by a linked-list of continuations. There is potential overlap between the evolution of the call-stacks for alternative parses. The Graph Structured Stack (GSS) is a data structure that efficiently represents multiple stacks simultaneously, by maximising sharing between stacks. A GSS is a (possibly cyclic) directed graph with continuations as nodes. All paths in a GSS end in the same sink, representing the empty stack. Each path represents the stack consisting of those continuations found at the nodes of the path. A GSS represents infinitely many stacks when it is cyclic, as infinitely many paths exist that traverse the cycle.

Scott and Johnstone's description of RGLL involves a modification of the GSS that embeds the original GSS, retaining the strong connection between the GSS and the call-stacks of RDP parsers [Scott and Johnstone, 2016]. In this variation, the GSS is a bipartite graph in which continuations are interspersed with a new type of node $\langle X, l \rangle$, with X a nonterminal and l an integer (left extent). We call such a pair a *commencement* (in contrast to continuation), as it identifies the call to the parse function of X with argument l. To construct a BSPPF, edges in the GSS are labelled with SPPF-nodes. Since we are constructing an extended packed node set, rather than a tree-structure, we have no need for this complication.

Unlike [Scott and Johnstone, 2016], we do not concern ourselves with maintaining a strong connection between the call-stacks in RDP and the call-graph we use. We model the call-graph as a binary relation $\mathcal{G}$ between commencements and continuations. A binary relation $\mathcal{P}$ between commencements and natural numbers models the storage of the zero or more right extents returned by a parse function, corresponding to the *pop-set* of GLL. Together, the relations reflect for each unique function call, identified by the commencement

$\langle X, l\rangle$, all the continuations C and the right extents R that have been discovered for $\langle X, l\rangle$ at a particular time of the algorithm's execution. The similarity with Johnson's memoisation combinator (see §3.3.2) is striking [Johnson, 1995].

2.4.3 Purely Functional GLL Parsing

We give a purely functional formalisation of a GLL algorithm, called FUN-GLL, operating on the abstract data structures introduced in previous sections as $\mathcal{G}$, $\mathcal{P}$, extended packed nodes, descriptors, and the worklist $\mathcal{R}$ of CDS-FORM. As such, we side-step a discussion on the implementation and efficiency of the data structures. Our claim is that, although the data structures are crucial for the efficiency and worst-case complexity of the algorithms, the specifics of the data structures' implementations are not crucial to the understanding of the algorithm. A thorough discussion of the data structures used by GLL algorithms is found in [Johnstone and Scott, 2011].

Formalisation

FUN-GLL differs only from CDS-FORM in that the set $\mathcal{U}$ is not inspected to determine whether a skip or ascend action can be performed. $\mathcal{G}$ and $\mathcal{P}$ are specialised data structures for retrieving that information. In CDS-FORM, $\mathcal{U}$ is inspected to find all descriptors of the form $\langle X ::= \alpha \cdot Y\beta, l, k\rangle$ whenever a descriptor $\langle Y ::= \delta\cdot, k, r\rangle$ is processed (the ascend action), whereas in FUN-GLL all continuations of the form $\langle X ::= \alpha Y\cdot\beta, l\rangle$ are recorded in $\mathcal{G}$ for commencement $\langle Y, k\rangle$ (δ is irrelevant). Similarly, in CDS-FORM, $\mathcal{U}$ is inspected to find all descriptors of the form $\langle Y ::= \delta\cdot, k, r\rangle$ whenever a descriptor of the form $\langle X ::= \alpha\cdot Y\beta, l, k\rangle$ is processed (the skip action), whereas in FUN-GLL all right extents r currently discovered are recorded in $\mathcal{P}$ for commencement $\langle Y, k\rangle$. In FUN-GLL, set $\mathcal{U}$ is only inspected to determine whether a certain descriptor has already been processed[8]. We present FUN-GLL as a variation on CDS-FORM. The differences between CDS-FORM and FUN-GLL are highlighted in blue.

Formalisation 2 (FUN-GLL)**.** Function *fungll* receives a grammar $\gamma = \langle Z, p\rangle \in \Gamma$ and a sentence $I \in T^*$ as input, and computes $\mathcal{U}(\gamma, I)$ and $\Delta(\mathcal{U}(\gamma, I))$.

[8] Because the algorithm only performs membership tests on $\mathcal{U}$, it can thus be implemented as a (three-dimensional) array with constant lookup.

$$\mathit{fungll}(\gamma, I) = \mathit{loop}_{\gamma,I}(\{\langle Z ::= \cdot\beta, 0, 0\rangle \mid Z ::= \beta \in \gamma\}, \emptyset, \emptyset, \emptyset, \emptyset)$$
$$\textbf{where } \langle Z, p\rangle = \gamma$$

In the definition of *loop* that follows, the call to *process* is given $\mathcal{G}$ and $\mathcal{P}$ as arguments, not $\mathcal{U}$, showing that $\mathcal{U}$ is not needed to process the item. Processing a descriptor might involve adding new entries to $\mathcal{G}$ and $\mathcal{P}$ in the recursive call of *loop* ($\mathcal{G}_1$ and $\mathcal{P}_1$).

$$\mathit{loop}_{\gamma,I}(\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E}) =$$
$$\begin{cases} \langle \mathcal{U}, \mathcal{E}\rangle & \textbf{if } \mathcal{R} = \emptyset \\ \mathit{loop}_{\gamma,I}(\mathcal{R} \cup \mathcal{R}_1 \setminus \mathcal{U}', \mathcal{U}', \mathcal{G} \cup \mathcal{G}_1, \mathcal{P} \cup \mathcal{P}_1, \mathcal{E} \cup \mathcal{E}_1) & \text{otherwise} \\ \quad \textbf{where } \langle\langle \mathcal{R}_1, \mathcal{E}_1\rangle, \mathcal{G}_1, \mathcal{P}_1\rangle = \mathit{process}_{\gamma,I}(d, \mathcal{G}, \mathcal{P}) & \\ \quad \textbf{and } \quad \mathcal{U}' = \{d\} \cup \mathcal{U} & \end{cases}$$
$$\textbf{where } d = \mathit{select}(\mathcal{R})$$

The definition of *process*, given below, shows that $\mathcal{G}$ is extended when a nonterminal is descended. $\mathcal{P}$ is extended when the end of a production is reached. The sets R and K are computed from the $\mathcal{P}$ and $\mathcal{G}$ respectively, rather than $\mathcal{U}$, as explained above. The functions *match*, *descend* and *skip* have not changed.

$$\mathit{process}_{\gamma,I}(\langle X ::= \alpha \cdot s\beta, l, k\rangle, \mathcal{G}, \mathcal{P}) =$$
$$\begin{cases} \langle \mathit{match}_{\gamma,I}(X ::= \alpha \cdot s\beta, l, k), \emptyset, \emptyset\rangle & \textbf{if } s \in T \\ \langle \mathit{descend}_{\gamma,I}(s, k), \{\langle\langle s, k\rangle, \langle X ::= \alpha s \cdot \beta, l\rangle\rangle\}, \emptyset\rangle & \textbf{if } R = \emptyset \\ \langle \mathit{skip}_{\gamma,I}(\langle X ::= \alpha s \cdot \beta, l, k\rangle, R), \{\langle\langle s, k\rangle, \langle X ::= \alpha s \cdot \beta, l\rangle\rangle\}, \emptyset\rangle & \text{otherwise} \end{cases}$$
$$\textbf{where } R = \{r \mid \langle\langle s, k\rangle, r\rangle \in \mathcal{P}\}$$
$$\mathit{process}_{\gamma,I}(\langle Y ::= \delta\cdot, k, r\rangle, \mathcal{U}, \mathcal{G}, \mathcal{P}) = \langle\langle \mathcal{R}_1, \mathcal{E}_1 \cup \mathcal{E}_2\rangle, \emptyset, \{\langle\langle Y, k\rangle, r\rangle\}\rangle$$
$$\textbf{where } \langle \mathcal{R}_1, \mathcal{E}_1\rangle = \mathit{ascend}(K, r)$$
$$\textbf{and } \quad \mathcal{E}_2 = \{\langle Y ::= \cdot, k, r, r\rangle \mid \delta = \epsilon\}$$
$$\textbf{and } \quad K = \{\langle X ::= \alpha Y \cdot \beta, l, k\rangle \mid \langle\langle Y, k\rangle, \langle X ::= \alpha Y \cdot \beta, l\rangle\rangle \in \mathcal{G}\}$$

Figure 2.8 shows an execution of FUN-GLL, processing the same number of descriptors as CDS-FORM to find a closed set of descriptors and extended packed nodes.

#	**processing**	**action**	**new**	$\mathcal{G}$	$\mathcal{P}$	$\mathcal{E}$
1	$\langle S ::= \cdot AA, 0, 0\rangle$	**descend**	$2, 3$	$(k_1, c_1\rangle$		
2	$\langle A ::= \cdot a, 0, 0\rangle$	**match**	4			e_1
3	$\langle A ::= \cdot E, 0, 0\rangle$	**descend**	5	(k_2, c_2)		
4	$\langle A ::= a\cdot, 0, 1\rangle$	**ascend**	6		$(k_1, 1)$	e_2
5	$\langle E ::= \cdot, 0, 0\rangle$	**ascend**	7		$(k_2, 0)$	e_9, e_3
6	$\langle S ::= A \cdot A, 0, 1\rangle$	**descend**	$8, 9$	(k_3, c_3)		
7	$\langle A ::= E\cdot, 0, 0\rangle$	**ascend**	10		$(k_1, 0)$	e_4
8	$\langle A ::= \cdot a, 1, 1\rangle$	**match**	-	-	-	-
9	$\langle A ::= \cdot E, 1, 1\rangle$	**descend**	11	(k_4, c_4)		
10	$\langle S ::= A \cdot A, 0, 0\rangle$	**skip**	$12, 13$	(k_1, c_3)		e_7, e_8
11	$\langle E ::= \cdot, 1, 1\rangle$	**ascend**	14		$(k_4, 1)$	e_{10}, e_5
12	$\langle S ::= AA\cdot, 0, 1\rangle$	**ascend**	-	-	-	-
13	$\langle S ::= AA\cdot, 0, 0\rangle$	**ascend**	-	-	-	-
14	$\langle A ::= E\cdot, 1, 1\rangle$	**ascend**	12!		$(k_3, 1)$	e_6

$$c_1 = \langle S ::= A \cdot A, 0\rangle \qquad k_1 = \langle A, 0\rangle$$
$$c_2 = \langle A ::= E\cdot, 0\rangle \qquad k_2 = \langle E, 0\rangle$$
$$c_3 = \langle S ::= AA\cdot, 0\rangle \qquad k_3 = \langle A, 1\rangle$$
$$c_4 = \langle A ::= E\cdot, 1\rangle \qquad k_4 = \langle E, 1\rangle$$

Figure 2.8: Execution of FUN-GLL.

Chapter 3

Generalising Combinator Parsing

Top down parsers implementing Recursive Descent (RD) parsers are written as a set of mutually recursive parse functions, one for each nonterminal of a grammar. As such, there is a strong correspondence between an RD parser and the grammar for which it has been written. Parser[1] combinators take this approach to a next level. Rather than (manually or mechanically) writing a parser based on a grammar, more complex parse functions are obtained by applying combinator functions to simpler parse functions. Fundamental to the parser combinator approach is that existing parsers are easily composed to form new parsers, relying on the ability to define higher-order functions in the host language. The approach enables the definition of combinators that capture common patterns like: arbitrarily delimited program fragments, program fragments being repeated possibly many times, or occurrences of left- or right-associative operators. The combinators can be used in multiple language definitions without modifications; they are executable and reusable components.

In this chapter we introduce the topic of combinator parsing by developing 'conventional' parser combinators in the style of [Wadler, 1985] and as found in many popular libraries across functional programming languages. Conventional implementations of parser

[1] Until we introduce semantic actions in Chapter 5, the term recogniser combinator is more accurate. We stick with established terminology however, and use the term 'parser' more liberally than in Chapter 2.

combinators have the same weaknesses as standard RD parsers; backtracking may result in exponential runtimes and parsers fail to terminate when they are defined with left-recursion. We discuss several approaches to overcoming these issues, focusing on two in particular. All of these approaches use a form of *observable sharing* to make recursive calls detectable. Some approaches use memoisation to prevent nontermination and the duplication of work generally, bringing down the runtime complexity.

This chapter lays the groundwork for the development of explicit BNF combinators in Chapter 4, the main contribution of this part of the thesis, bringing combinator parsing and generalised parsing together.

3.1 Combinator Parsing

To introduce our approach to combinator parsing, we first present a conventional parser combinator library in this section. In subsequent sections, we consider alternative definitions of the same combinators in order to explain ways to generalise the combinator approach for overcoming the drawbacks of the original definitions.

3.1.1 Elementary Combinators

Throughout this chapter we discuss several types of parse functions. A parse function typically receives as arguments an input sentence I and an index into the sentence (left extent l) and returns another index (right extent r). A sentence is a sequence of terminal symbols taken from the set W, i.e. $I \in W^*$. A parser combinator library provides a number of elementary parse functions, functions generating elementary parse functions, elementary combinators, and derived combinators. Together these are (somewhat confusingly) referred to as parser combinators. A parser combinator is called *derived* if it is defined in terms of already existing parser combinators, otherwise the combinator is *elementary*. We refer to an expression formed out of combinators as a *combinator expression*.

A typical parser combinator library provides implementations of a similar set of core combinators and generators. Here we consider these to be: the parse function generator *term*, elementary parse functions *fails* and *succeeds*, and elementary combinators *seq* and *alt*. Figure 3.1 gives a standard definition to these combinators based on a backtracking RD

$$
\begin{aligned}
term(x)(I,i) &= \begin{cases} \{i+1\} & \textbf{if } I_i = x \\ \emptyset & \text{otherwise} \end{cases} \\
seq(p,q)(I,i) &= \{r \mid k \in p(I,i), r \in q(I,k)\} \\
alt(p,q)(I,i) &= p(I,i) \cup q(I,i) \\
succeeds(I,i) &= \{i\} \\
fails(I,i) &= \emptyset \\
recognise(p)(I) &= \begin{cases} \textbf{true} & \textbf{if } |I| \in p(I,0) \\ \textbf{false} & \text{otherwise} \end{cases}
\end{aligned}
\qquad
\begin{aligned}
term &: W \to \mathbf{M}_1 \\
seq &: \mathbf{M}_1 \times \mathbf{M}_1 \to \mathbf{M}_1 \\
alt &: \mathbf{M}_1 \times \mathbf{M}_1 \to \mathbf{M}_1 \\
succeeds &: \mathbf{M}_1 \\
fails &: \mathbf{M}_1 \\
recognise &: \mathbf{M}_1 \to (W^* \to \mathbb{B}) \\
\textbf{where } \mathbf{M}_1 &= W^* \times \mathbb{N} \to \mathcal{P}(\mathbb{N})
\end{aligned}
$$

Figure 3.1: Backtracking RD implementation of the conventional elementary combinators, together with *recognise* for running a parse function.

algorithm[2]. In these definitions, the set $\mathbf{M}_1$ of all parse functions is defined as $W^* \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$. Function *term* generates a parse function that matches and consumes the given terminal symbol and no other terminals. Function *seq* 'runs' its first operand to get zero or more right extents and then 'runs' its second argument with those right extents as the index argument. Function *alt* generates a parse function that 'runs' both of its operands and combines their result. Function *succeeds* consumes no input but always succeeds, whilst *fails* consumes no input but always fails. Function *recognise* transforms a parse function p into a function from an input sentence I to a Boolean by applying p to I and 0, the initial left extent, and tests whether the length of I is part of the result. Function $recognise(p)$ is a recogniser for the language $\mathcal{L}(\gamma)$ if $I \in \mathcal{L}(\gamma) \iff recognise(p)(I) = \textbf{true}$. Section §3.2.3 discusses the possibility of generating, from a combinator expression, a grammar γ for which this holds.

3.1.2 Efficiency and Termination

Parsers are written in a host language that offers a mechanism for associating identifiers with expressions. This enables recursive parsers that recognise infinitely many input sentences. The mechanism also enables simple definitions of otherwise complicated parsers via reuse.

The *alt* combinator requires full evaluation of both its operands, which, in the combina-

[2]This is the first time we define higher-order functions, for which we use several layers of parenthesised parameters. For example, the 'curried' version of a function with three parameters is defined as $f(a)(b)(c) = ...$ and applied as $f(1)(2)(3)$. The uncurried version is defined as $f(a,b,c) = ...$ and applied as $f(1,2,3)$.

```
pX = alt (term ('a')
         , alt (seq (term ('a'), pX)
                , seq (term ('a'), seq (term ('a'), pX))))
```

Figure 3.2: A parser with exponential runtime.

```
pY = alt (term ('a')
         , alt (seq (term ('a'), term ('a'))
                , seq (pY, pY)))
```

Figure 3.3: A parser failing to terminate on the empty sentence.

tion with named expressions, potentially causes repeated work and nontermination. In other words, all branches introduced by *alt* are explored. Figure 3.2 shows a parser for which an input sentence consisting of n terminals[3] `'a'` can be recognised in exponentially-in-n many ways[4]. Evaluating $pX(I, 0)$ requires exponentially many reductions[5] in the length of I, if I consists only of `'a'`s. Figure 3.3 defines pY, which fails to terminate on any sentence[5], as each call to pY can result in a call to pY without having consumed any input, i.e. the left extent is unchanged. It is well known that standard RD parsers fail to terminate on left-recursive grammars [Grune, 2010, Wadler, 1985] and the problem mentioned here is closely related. This triggers the question whether, like GLL does for RD, parser combinators can be generalised to allow for left-recursion. In later sections we discuss approaches to generalising the parser combinator approach so that parsers terminate efficiently for larger classes of grammars.

3.1.3 Combinator Laws

Figure 3.4 presents a number of laws that hold for the parser combinators in Figure 3.1. From these laws it follows that $\mathbf{M}_1$ forms a semi-ring with *seq* as its multiplication operation and *alt* as its addition operation with respective identities *succeeds* and *fails*. The laws enable

[3] In our examples we choose W to be the set of characters, denoted by single quotation.
[4] The number of ways in which the string can be recognised grows like the Fibonacci sequence.
[5] In a strict (applicative-order) evaluation model.

$$
\begin{aligned}
alt(fails, q) &= q && \text{(left-identity-alt)}\\
alt(p, fails) &= p && \text{(right-identity-alt)}\\
alt(p, p) &= p && \text{(idempotency-alt)}\\
alt(p, q) &= alt(q, p) && \text{(commutativity-alt)}\\
alt(p, alt(q, r)) &= alt(alt(p, q), r) && \text{(associativity-alt)}\\
seq(succeeds, q) &= q && \text{(left-identity-seq)}\\
seq(p, succeeds) &= p && \text{(right-identity-seq)}\\
seq(fails, q) &= fails && \text{(left-absorption-seq)}\\
seq(p, fails) &= fails && \text{(right-absorption-seq)}\\
seq(p, seq(q, r)) &= seq(seq(p, q), r) && \text{(associativity-seq)}\\
seq(p, alt(q, r)) &= alt(seq(p, q), seq(p, r)) && \text{(left-distributivity)}\\
seq(alt(p, q), r) &= alt(seq(p, r), seq(q, r)) && \text{(right-distributivity)}
\end{aligned}
$$

Figure 3.4: Properties maintained by the combinators definitions of Figure 3.1.

high-level reasoning about combinator expressions. For example, we can conclude that the ordering of alternatives does not matter (commutativity), that duplicate alternates can be removed (idempotency) and that we can perform left-factoring to avoid running a parse functions multiple times (left-distributivity).

3.1.4 Extensions

The combinators *term*, *seq*, *succeeds*, *alt* and *fails* form the core of a basic combinator library. The library can be extended by defining derived combinators and additional elementary combinators.

Derived combinators As an example of a derived combinators, consider *within*:

$$
\begin{aligned}
&within : \mathbf{M}_1 \times \mathbf{M}_1 \times \mathbf{M}_1 \to \mathbf{M}_1\\
&within(p, m, q) = seq(p, seq(m, q))
\end{aligned}
$$

This useful combinator can be specialised to form a combinator that places a given parse function in between parse functions matching opening and closing parentheses.

$$
\begin{aligned}
&\mathit{parens} : \mathbf{M}_1 \to \mathbf{M}_1 \\
&\mathit{parens}(p) = \mathit{within}(\mathit{term}(\texttt{'('}), p, \mathit{term}(\texttt{')'}))
\end{aligned}
$$

The combinator *sepBy* is given two parse functions, and forms a parser that applies the first argument one or more times, each time separated by a separator determined by the second argument.

$$
\begin{aligned}
&\mathit{sepBy} : (\mathbf{M}_1 \times \mathbf{M}_1) \to \mathbf{M}_1 \\
&\mathit{sepBy}(p, s) = \mathit{alt}(p, \mathit{seq}(p, \mathit{seq}(s, \mathit{sepBy}(p, s))))
\end{aligned}
$$

Together, the last two derived combinators can be used to define a derived combinator for matching programming language tuples containing expressions, i.e. $\mathit{tuples}(p) = \mathit{parens}(\mathit{sepBy}(p, \texttt{','}))$, if p is a parse function for matching expressions.

Based on the laws in Figure 3.4 we can refactor the definition of *sepBy* to make it more efficient. The underlying observation is that, in the definition above, p is applied twice, one time at the start of each of the alternatives. The following equalities show how the original definition of *sepBy* can be changed to a more efficient version using the left-distributivity law.

$$
\begin{aligned}
&\mathit{alt}(p, \mathit{seq}(p, \mathit{seq}(s, \mathit{sepBy}(p, s)))) && \text{(right-identity-seq)} \\
=&\mathit{alt}(\mathit{seq}(p, \mathit{succeeds}), \mathit{seq}(p, \mathit{seq}(s, \mathit{sepBy}(p, s)))) && \text{(left-distributivity)} \\
=&\mathit{seq}(p, \mathit{alt}(\mathit{succeeds}, \mathit{seq}(s, \mathit{sepBy}(p, s))))
\end{aligned}
$$

Combinators can be higher-order in the sense of receiving a combinator, rather than a parse function, as an argument. The combinator *flatSeq* defined below produces a parse function that matches delimited sequences with a separator, where the elements of the sequence may be delimited sequences of the same form. By abstracting over the delimiter we obtain a higher-order combinator.

$$
\begin{aligned}
&\mathit{flatSeq} : ((\mathbf{M}_1 \to \mathbf{M}_1) \times \mathbf{M}_1 \times \mathbf{M}_1) \to \mathbf{M}_1 \\
&\mathit{flatSeq}(d, p, s) = \mathit{alt}(p, d(\mathit{sepBy}(\mathit{flatSeq}(d, p, s), s)))
\end{aligned}
$$

(Comparing this definition with the definition of *sepBy* itself is insightful.) The parse function $\mathit{flatSeq}(\mathit{parens}, \mathit{term}(\texttt{'a'}), \mathit{term}(\texttt{','}))$ recognises the strings `"a"`, `"(a)"`, `"((a))"`, `"(a,a)"`, `"(a,(a,a))"`, `"(a,(a,a),(a))"`, etc.

Additional elementary combinators As an example of an additional elementary combinator, consider *pred*, a generalisation of *term*, which matches terminal symbols that satisfy a certain predicate:

$$pred : (W \to \mathbb{B}) \to \mathbf{M}_1$$
$$pred(f)(I, i) = \begin{cases} \{i + 1\} & \textbf{if } f(I_i) \\ \emptyset & \text{otherwise} \end{cases}$$

Another useful elementary combinator checks whether a parse functions succeeds without consuming input:

$$peek : \mathbf{M}_1 \to \mathbf{M}_1$$
$$peek(p)(I, i) = \begin{cases} \{i\} & \textbf{if } p(I, i) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Combinator *not* constructs a parse function that fails if the given parse function is successful.

$$not : \mathbf{M}_1 \to \mathbf{M}_1$$
$$not(p)(I, i) = \begin{cases} \emptyset & \textbf{if } p(I, i) \neq \emptyset \\ \{i\} & \text{otherwise} \end{cases}$$

This combinator can be used to enforce a "longest-match" strategy, for example to greedily match as many digits as possible:

$$digit = alt(term(\texttt{'0'}), alt(term(\texttt{'1'}), alt(\ldots, alt(term(\texttt{'8'}), term(\texttt{'9'}))) \ldots))$$
$$digits = alt(not(digit), seq(digit, digits))$$

3.1.5 Discussion

This section has shown how combinators can be implemented as recognisers, how identifiers can be associated with combinator expressions to define succinct and cyclic parsers, and that such parsers may be inefficient or fail to terminate entirely.

Several authors have suggested combinator implementations that ensure that parsers are guaranteed to terminate within a polynomial order of steps (with respect to the length of the input sentence). The efficiency of the parsers can be improved by memoisation, as suggested by [Norvig, 1991]. To handle left-recursion, Frost, Hafiz and Callaghan add a 'curtailment' procedure to memoising combinators with the effect of limiting the number of recursive calls based on the length of the input sentence [Frost et al., 2008]. [Johnson, 1995]

gives an elegant implementation of memoising combinators that store not only parse results (right extents) but also continuation functions. The stored continuations are used to detect recursion as well as to ensure that all possible derivation paths are explored.

A different approach was separately suggested by [Ljunglöf, 2002], [Ridge, 2014], and [Devriese and Piessens, 2012]. Their approaches have in common that they generate a grammar from a combinator expression and then apply a standalone parser to find one or more derivations. A final process applies semantic actions based on the derivation(s).

Each of these approaches relies on 'observable sharing', a concept studied in the context of embedding DSLs in purely-functional languages [Claessen and Sands, 1999, Gill, 2009]. A nice introduction to observable sharing is given by [Gill, 2009], discussing several methods via which observable sharing can be achieved. Baars and Swierstra discuss observable sharing in the context of parser combinators [Baars and Swierstra, 2004]. The *inject* combinator, introduced in the next section, makes sharing observable by relying on the programmer to associate unique labels with sharing positions. A combinator like *inject* is often seen as a non-solution to observable sharing, as making sharing observable becomes the responsibility of the programmer. However, in the context of this thesis we think it is both pragmatic and practical, as demonstrated by the case studies of this thesis, and the work of others [Ridge, 2014, Frost et al., 2008]. The main practical concern is that, as a program grows over time, it becomes harder to guarantee that the labels are unique.

3.2 Generating a Binarised Grammar

[Ridge, 2014] introduces the combinator library P3 that is implemented so that the evaluation of combinator expressions involves three phases. Firstly, a grammar is extracted from a combinator expression. Secondly, the grammar is given, together with an input sentence, to a parsing procedure (an implementation of Earley's algorithm) to obtain an 'oracle' representing all possible derivations of the input sentence according to the grammar. Thirdly, the combinator expression is traversed a second time, in an RD fashion, parsing the input sentence under the guidance of the oracle whilst applying semantic actions.

The second phase can be based on any parsing procedure that produces an oracle. In §5.3.2, a variant of Ridge's third phase (or 'semantic phase') is implemented in which a set

$$
\begin{aligned}
term_R(x) &= \langle nt(x), \{nt(x) ::= x\}\rangle \\
seq_R(\langle p_1, p_2\rangle, \langle q_1, q_2\rangle) &= \langle p_1 \times_{+\!\!+} q_1, \{(p_1 \times_{+\!\!+} q_1) ::= p_1 q_1\} \cup p_2 \cup q_2\rangle \\
alt_R(\langle p_1, p_2\rangle, \langle q_1, q_2\rangle) &= \langle p_1 +_{+\!\!+} q_1, \{(p_1 +_{+\!\!+} q_1) ::= p_1, (p_1 +_{+\!\!+} q_1) ::= q_1\} \cup p_2 \cup q_2\rangle
\end{aligned}
$$

Figure 3.5: Implementation of *term*, *seq* and *alt* computing a binarised grammar.

of extended packed nodes is used as an oracle, thus enabling FUN-GLL for use in the second phase. In this section we explain Ridge's first phase. An approach similar to Ridge's was developed independently by [Ljunglöf, 2002].

3.2.1 Basic Grammar Generation

The goal is to generate a grammar γ by evaluating a combinator expression e. This is achieved by generating a string for each combinator expression based on the strings generated for its subexpressions, thus capturing the precise structure of the expression. The string is used as a nonterminal symbol in the generated grammar. We assume an injective function nt from W to strings. Furthermore, we define the following operators for combining strings (where $+\!\!+$ is string concatenation):

$$
\begin{aligned}
p \times_{+\!\!+} q &= \texttt{"*("} +\!\!+ p +\!\!+ \texttt{","} +\!\!+ q +\!\!+ \texttt{")"} \\
p +_{+\!\!+} q &= \texttt{"+("} +\!\!+ p +\!\!+ \texttt{","} +\!\!+ q +\!\!+ \texttt{")"}
\end{aligned}
$$

With these functions we define $term_R$, seq_R and alt_R in Figure 3.5. The terminal symbols of the generated grammars are elements of W, whereas the nonterminals are strings. The grammars generated this way are binarised in the sense that each nonterminal has one production with at most two symbols or two productions with at most one symbol.

This definition of the combinators is problematic as recursive combinator expressions cannot be reduced in a finite amount of steps. Consider a recursively defined combinator expression $e = alt(term(\texttt{'a'}), e)$. The definition of *alt* tells us that $fst(e)$, the first component of e, is $nt(\texttt{'a'}) +_{+\!\!+} fst(e)$, a contradiction. We use $inject_R$ to 'break the cycle'[6].

$$
inject_R(l, \langle p_1, p_2\rangle) = \langle show(l), p_2\rangle
$$

[6]In Ridge's work, the equivalent combinator is called *mkntparser*.

The first argument of $inject_R$ is an element $l \in L$, where L is a set of labels, and *show* is an injective function from labels to strings. From the definition of $inject_R$ we can directly conclude[7] that the first component of $e' = inject_R(\texttt{"E"}, alt_R(term_R(\texttt{'a'}), e'))$ is `"E"`. To find the second component of e' we use the observation that identical subexpressions have the same 'contribution' to the set of productions of the generated grammar. If recursive calls can be observed, we can decide not to add any productions whenever a combinator expression is evaluated as part of a recursive call. If $inject_R$ is applied to recursive expressions, then the recursion can be observed by propagating a set of injected labels, as we shall demonstrate in the next section. In §3.1.4 we saw the combinator *sepBy* and parse function *digits*, which are both defined using recursion.

3.2.2 Grammar Generation with Observable Sharing

If we assume that all recursively defined combinator expressions are wrapped in an occurrence of $inject_R$, then we can use the inserted label to observe the recursion. Doing so, we give an alternative definition of $term_R$, seq_R, alt_R and $inject_R$ in Figure 3.6. The second component of a combinator expression is no longer a set of productions, but a function that given a set of labels C and a set of productions P returns a (possibly extended) set of labels $C' \supseteq C$ and a (possibly extended) set of productions $P' \supseteq P$. C holds all the injected labels encountered so far, whilst P holds all the productions of the computed grammar discovered so far[8]. The grammar generated for a particular expression $e = \langle n, gen \rangle$ is $\langle n, snd(gen(\emptyset, \emptyset)) \rangle$, as captured by the definition of *grammarOf*, where $snd(x)$ gives the second component of x.

Function $inject_R$ performs a form of memoisation on the second component of its second argument, and is useful to indicate not only recursive positions but sharing generally. The introduction of $inject_R$ has provided the tools required to:

1. guarantee termination by applying $inject_R$ to recursive expressions
2. prevent repeated evaluation by applying $inject_R$ to associate identical labels with equivalent subexpressions[9]

[7] Here we use strings for labels, thus *show* is the identity function on strings.
[8] The propagation of C and P determines a left-to-right traversal.
[9] As Ridge notes, two equal expressions generate the same productions automatically.

$$
\begin{aligned}
term_R(x) &= \langle nt(x), gen\rangle \\
\textbf{where } gen(C,P) &= \langle C, \{nt(x) ::= x\} \cup P\rangle \\
seq_R(\langle p_1,p_2\rangle,\langle q_1,q_2\rangle) &= \langle p_1 \times_{\#} q_1, gen\rangle \\
\textbf{where } gen(C,P) &= (q_2 \circ p_2)(C, \{(p_1 \times_{\#} q_1) ::= p_1 q_1\} \cup P) \\
alt_R(\langle p_1,p_2\rangle,\langle q_1,q_2\rangle) &= \langle p_1 +_{\#} q_1, gen\rangle \\
\textbf{where } gen(C,P) &= (q_2 \circ p_2)(C, \{(p_1 +_{\#} q_1) ::= p_1, (p_1 +_{\#} q_1) ::= q_1\} \cup P) \\
inject_R(x,\langle p_1,p_2\rangle) &= \langle show(x), gen\rangle \\
\textbf{where } gen(C,P) &= \begin{cases} \langle C,P\rangle & \textbf{if } x \in C \\ p_2(C \cup \{x\}, P \cup \{show(x) ::= p_1\}) & \text{otherwise} \end{cases} \\
grammarOf(\langle n,p\rangle) &= \langle n, snd(p(\emptyset,\emptyset))\rangle
\end{aligned}
$$

Figure 3.6: Alternative to Figure 3.5, sensitive to injected symbols.

The introduction of $inject_R$ does not rule out that nonterminating and costly-to-evaluate expressions are written, as one may fail to use $inject_R$ where necessary. Moreover, compared to conventional parser combinators, this set of elementary combinators is harder to extend, as explained in the next subsection.

3.2.3 Limitations

Recursive parameterised combinators With conventional combinators, as defined in §3.1.1, it is possible to define new combinators that receive combinator expressions as arguments. As an example we have seen the derived combinator *within*. Since *within* is not recursive, it can be defined in terms of seq_R and $term_R$ just as easily. However, the recursively defined derived combinator *sepBy* is more challenging to redefine. This requires using $inject_R$ in such a way that a different symbol is injected when different arguments are given to *sepBy*. For example, we expect $sepBy(p, term(\texttt{'.'}))$ and $sepBy(p, term(\texttt{','}))$ to generate different nonterminals. In other words, the first argument of $inject_R$ must somehow be based on the arguments of *sepBy*.

This problem is more acute when a combinator is defined recursively with an argument that changes endlessly. The function *scales* is an example of such a combinator (*parens* is easily redefined).

$$scales(p) = alt(p, seq(p, scales(parens(p))))$$

Function $recognise(scales(term(\texttt{'a'})))$ terminates on all input sentences and recognises the infinite language $\{\texttt{"a"}, \texttt{"a(a)"}, \texttt{"a(a)((a))"}, \texttt{"a(a)((a))(((a)))"}, \ldots\}$. It is not clear how *scales* can be defined as a grammar generating combinator.

Another problem is that not all arguments can be converted into strings. A higher-order combinator like *flatSeq*, discussed in §3.1.4, receives a combinator as an argument, which cannot directly be converted to a (unique) string.

Additional elementary combinators Given that one understands the parsing algorithm underlying the definitions of *term*, *seq*, *alt*, *fails* and *succeeds*, it is quite easy to add additional elementary combinators, for which we have used *pred*, *peek* and *not* as examples. So far, the set of elementary combinators for generating grammars consists only of $term_R$, seq_R and alt_R. To define $succeeds_R$ we can nominate a terminal symbol $s^0 \in W$ that represents the empty sequence of symbols. This symbol should be reserved in the sense that it cannot appear in any input sentence nor as an argument to $term_R$. We can then define $succeeds_R$ as follows.

$$succeeds_R = \langle nt(s^0), gen\rangle \quad \textbf{where } gen(C, P) = \langle C, \{nt(s^0) ::= \epsilon)\} \cup P\rangle$$

Similarly we can reserve a symbol $s^\perp \in W$ and use it to define $fails_R$.

$$fails_R = \langle nt(s^\perp), gen\rangle \quad \textbf{where } gen(C, P) = \langle C, \{nt(s^\perp) ::= s^\perp\} \cup P\rangle$$

The production $nt(s^\perp) ::= s^\perp$ will never be used in a derivation since $s^\perp$ is assumed not to occur in any input sentence.

We have thus introduced special symbols that occur only in productions and not in input sentences. It is not clear how elementary combinators like *pred*, *peek* and *not* can be defined without such special symbols or without extending the notion of productions and grammars.

Combinator laws The laws from Figure 3.4 do not hold for the grammar generating combinators. For example, $succeeds_R$ is not an identity of seq_R, as the presence of $succeeds_R$ adds a production for the nonterminal $nt(s^0)$ to the generated grammar, as well as a production $(nt(s^0) \times_{\shortparallel} fst(q)) ::= nt(s^0)fst(q)$. However, the grammars are equivalent under

certain interpretations. Similarly, alt_R and seq_R are no longer associative. This may not be a problem when such details of the grammar are irrelevant to the application. However, library implementers cannot use the laws to justify refactorings when they do not wish to make assumptions about the usage of their library.

Generating hash-codes It may be possible to generate grammars such that the laws are upheld, if the nonterminals are generated as hash-codes rather than strings. This would likely require an associative and commutative binary hash-function for composing the hash-codes produced for the operands of *alt*, an associative and non-commutative binary hash-function for composing the hash-codes produced for the operands of *seq* and a non-associative hash-function for combining the label of a recursive combinator with parameters and the hash-codes of its arguments.

3.3 Memoising Continuation-passing Combinators

In [Johnson, 1995], Johnson gives an elegant implementation in SCHEME of memoising combinators that store not only parse results (right extents) but also continuation functions. The stored continuation functions are used to detect recursion as well as to ensure that all possible paths are explored (continuation management, related to §2.4.1). The stored right-extents are used for classical memoisation, shortcutting certain execution paths and avoiding repeated work. This section gives a purely functional formalisation of Johnson's approach using *inject*. The first step is to explain a basic definition of the elementary combinators written in continuation-passing style.

3.3.1 Continuation-passing Recogniser

Section 2.4 explains that GLL parsing generalises standard recursive descent parsing by managing continuations explicitly so as to avoid relying on the host language's call-stack. Continuations are stored in a data structure — the GSS — and are applied as many times as necessary. [Johnson, 1995] takes a similar approach, implementing combinators for recognition. In our formalisation of GLL, the continuations are return positions paired with a left extent. In Johnson's combinators, continuations are parse functions — we refer to them

$$
\begin{aligned}
term(x)(c)(I,i) &= \begin{cases} c(i+1) & \textbf{if } I_i = x \\ \emptyset & \text{otherwise} \end{cases} \\
seq(p,q)(c_0)(I,i) &= p(c_1)(I,i) \\
&\qquad \textbf{where } c_1(r) = q(c_0)(I,r) \\
alt(p,q)(c)(I,i) &= p(c)(I,i) \cup q(c)(I,i) \\
succeeds(c) &= c \\
fails(c)(I,i) &= \emptyset \\
recognise(p)(I) &= \begin{cases} \textbf{true} & \textbf{if } |I| \in p(\overline{c})(I,0) \quad \textbf{where } \overline{c}(r) = \{r\} \\ \textbf{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.7: RD implementation in continuation-passing style.

as continuation functions — that when applied to a right extent 'do the remaining work' of matching the input string. The continuation functions are stored in a memoisation table so that they can be applied as many times as necessary. Our goal is to develop, in §3.3.2, a collection of combinators in the style of Johnson based on FUN-GLL. As a lead-in, we give a variation on the combinators of Figure 3.1, written in continuation-passing style (CPS).

As before, a parse function receives a fixed input string and an index as arguments and returns a set of right extents. A continuation function is a parse function without the input string argument. A combinator expression is no longer a parse function, but a function from continuation functions to parse functions. The set of all parse functions $\mathbf{M}_1$ is defined as $S^* \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$ (as in §3.1.1), the set of all continuation functions C_p is defined as $\mathbb{N} \to \mathcal{P}(\mathbb{N})$. A combinator expression is thus an element of $C_p \to F_p$. The definitions of the elementary combinators are give in Figure 3.7.

In this implementation of the combinators, the nondeterminism of *alt* is 'handled' by giving a copy of the current continuation function c to both alternates of *alt*. The same continuation function may thus be applied many times. The results of all parse functions are united (whether successful or not).

3.3.2 Memoising Combinators with Observable Sharing

Johnson defines a combinator *memo* that is applied to a combinator expressions and evaluates to a closure containing the combinator expression and a (unique) reference to a memoisation table. The reference is used to observe recursion and sharing generally. Instead of the impure *memo* combinator of [Johnson, 1995], we use *inject*, forcing the user of the combinator library — rather than the SCHEME interpreter — to provide a unique label to be associated with a particular combinator expression. We proceed by giving another implementation to the elementary combinators *term*, *seq*, *alt*, *inject*, *succeeds* and *fails*. The resulting combinator definitions are a purely functional formalisation of a variation on Johnson's combinators. The differences between this formalisation and Johnson's implementation are that we use sets[10] to hold continuation functions, rather than lists, and that we use a different method for observable sharing, influencing the user-experience.

A memo-table M is a mapping from a label and extent pair $\langle x, i\rangle$ to a set of right extents R and a set of continuation functions C. The functions *addToR* and *addToC* add a right extent r and a continuation function c to a memo-table respectively[11].

$$
\begin{aligned}
&\mathit{addToR}(x, i, r, M) = M' \\
&\quad \textbf{where } M'(x', i') = \langle R \cup \{r \mid x' = x, i' = i\}, C\rangle \\
&\quad \textbf{and } \quad \langle R, C\rangle = M(x, i) \\
&\mathit{addToC}(x, i, c, M) = M' \\
&\quad \textbf{where } M'(x', i') = \langle R, C \cup \{c \mid x' = x, i' = i\}\rangle \\
&\quad \textbf{and } \quad \langle R, C\rangle = M(x, i)
\end{aligned}
$$

The output of a parse function is a function from a memo-table to an updated memo-table and a set of right extents. To combine the outputs $\psi \in \Psi$ of several parse functions, whilst propagating the memo-table, the operator $\fatsemi$ is introduced:

$$
\begin{aligned}
&\fatsemi : \Psi \to \Psi \to \Psi \\
&(\psi_1 \fatsemi \psi_2)(M_0) = \langle M_2, R_1 \cup R_2\rangle \\
&\quad \textbf{where } \langle M_1, R_1\rangle = \psi_1(M_0) \\
&\quad \textbf{and } \quad \langle M_2, R_2\rangle = \psi_2(M_1)
\end{aligned}
$$

[10] See also footnote 11.

[11] Function *addToC* may be difficult to implement since adding continuation functions to a set requires the ability to compare continuation functions. Perhaps for this reason, Johnson uses lists in his formulation in SCHEME. As discussed later, using lists increases the worst-case runtime complexity.

There are two situations in which the information in a memo-table is used. Firstly, when there is a subsequent call to a memoised combinator expression, recognised by the availability of continuation functions in the table. In this case, the current continuation function c is to be applied to all the right extents in the table. This is done by the function *appToR*:

$$appToR(x, i, c, M) = {}^{\circ}_{9}\{c(r) \mid M(x, i) = \langle R, C\rangle, r \in R\}(M)$$

Here ${}^{\circ}_{9}\{\ldots\}$ is the generalisation of ${}^{\circ}_{9}$ that reduces a set[12] $\{\psi_1, \ldots, \psi_k\}$ to a single ψ' such that $\psi' = {}^{\circ}_{9}\{\psi_1, \ldots, \psi_k\} = \psi_1 \,{}^{\circ}_{9} \ldots {}^{\circ}_{9}\, \psi_k \,{}^{\circ}_{9}\, \psi_0$, with $\psi_0(M) = \langle M, \emptyset\rangle$. Note that, as before, continuation functions are parse functions without an input string as argument.

Secondly, a continuation function is interrupted when it is applied to a right extent that has already been encountered. If the right extent is new then all the continuation functions currently stored in the table are to be applied to it. The latter is done by *appToC*:

$$appToC(x, i, r, M) = {}^{\circ}_{9}\{c(r) \mid M(x, i) = \langle R, C\rangle, c \in C\}(M)$$

These continuation functions are specific to a particular label x and left extent i and are generated by the function *contFor* as follows:

$$contFor(x, i)(r)(M) = \begin{cases} \langle M, \emptyset\rangle & \textbf{if } M(x, i) = \langle R, C\rangle, r \in R \\ appToC(x, i, r, M') & \text{otherwise} \end{cases}$$
$$\textbf{where } M' = addToR(x, i, r, M)$$

Note the symmetry between *appToR* and *appToC* and the correspondence to the similarly symmetrical *skip* and *ascend* functions of FUN-GLL in §2.4.3.

A combinator expression is memoised by an application of *inject*, defined as follows:

$$inject(x, p)(c)(I, i)(M) = \begin{cases} \langle M, \emptyset\rangle & \textbf{if } c \in C \\ appToR(x, i, c, M') & \textbf{if } C \neq \emptyset \textbf{ and } c \notin C \\ p(contFor(x, i))(I, i)(M') & \text{otherwise} \end{cases}$$
$$\textbf{where } \langle R, C\rangle = M(x, i)$$
$$\textbf{and} \quad M' = addToC(x, i, c, M)$$

The core combinators are defined in Figure 3.8.

Function *recognise* transforms a combinator expression p into a recogniser by applying it to the base continuation function $\overline{c}(r)(M) = \langle M, \{r\}\rangle$, input string I, index 0, and the

[12] The order in which the parse functions are executed may influence efficiency, not the outcome.

$$
\begin{aligned}
term(x)(c)(I,i)(M) &= \begin{cases} c(i+1)(M) & \textbf{if } I_i = x \\ \langle M, \emptyset \rangle & \text{otherwise} \end{cases} \\
seq(p,q)(c_0)(I,l) &= p(c_1)(I,l) \\
&\quad \textbf{where } c_1(r) = q(c_0)(I,r) \\
alt(p,q)(c)(I,i) &= p(c)(I,i) \fatsemi q(c)(I,i) \\
succeeds(c)(I,i) &= c(i) \\
fails(c)(I,i)(M) &= \langle M, \emptyset \rangle \\
recognise(p)(I) &= \begin{cases} \textbf{true} & \textbf{if } |I| \in R \\ & \textbf{where } \langle M', R \rangle = p(\overline{c})(I,0)(\overline{M}) \\ & \textbf{ and } \overline{c}(r)(M) = \langle M, \{r\} \rangle \\ & \textbf{ and } \overline{M}(x,i) = \langle \emptyset, \emptyset \rangle \\ \textbf{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.8: Generalised continuation-passing combinators.

empty memo-table $\overline{M}(x,i) = \langle \emptyset, \emptyset \rangle$. The string I is recognised if its length is in the produced set of right extents R.

The striking similarity between the algorithm underlying the combinators of this section and FUN-GLL suggest that it is possible to define combinators based on FUN-GLL, and we do so in Section 4.3. In Section 4.3 we also show that combinators can compute grammar slots and use these to find extended packed nodes.

3.3.3 Duplicated Continuation Functions

Besides the base continuation function $\overline{c}$, there are two positions where new continuation functions are created: as part of *inject*, with the help of *contFor*, and as part of *seq*. In the former case, continuation functions are memoised so that a second call with the same right extent terminates directly. In the latter case, continuation functions are not memoised and may result in duplication of work. Let p be a combinator expression that matches the first n symbols in input sentence I and let q match the first m symbols, i.e. $p(c)(I,0) = c(n)$ and $q(c')(I,0) = c'(m)$ for all continuation functions c and c'. The following equations show that the potentially costly and unmemoised continuation function $\hat{c}$ may be used twice in an application of *inject*.

$$
\begin{aligned}
& seq(alt(p,q), inject(l,\ldots))(\hat{c})(I,0) \\
=& alt(p,q)(c_1)(I,0) && (\textbf{where } c_1(r) = inject(l,\ldots)(\hat{c})(I,r)) \\
=& p(c_1)(I,0) \fatsemi q(c_1)(I,0) \\
=& c_1(n) \fatsemi c_1(m) \\
=& inject(l,\ldots)(\hat{c})(I,n) \fatsemi inject(l,\ldots)(\hat{c})(I,m)
\end{aligned}
$$

In Johnson's SCHEME implementation, memo-table entries hold a list of continuation functions rather than a set[13], meaning that the same continuation function may be stored multiple times in the same entry (when $n = m$ above), assuming that the condition $c \in C$ of *inject* is not implemented (e.g. for the same reason that lists are used instead of sets).

This has been observed by Izmaylova, Afroozeh and Van der Storm in their discussion of [Johnson, 1995], on which their Meerkat library is based [Izmaylova et al., 2016]. By performing classical memoisation on all continuation functions, they reduce the runtime complexity from $O(n^{m+1})$ to $O(n^3)$ where n is the size of the input sentence and m is the length of the longest rule[14] in the grammar.

The issue discussed here can be explained in comparison with FUN-GLL of §2.4.3. In §2.4.3, continuations are modelled as a pair of a grammar slot and a left extent and can be seen to uniquely identify the next recursive descent parse function to be executed, or to continue its execution, corresponding to continuation functions in this section. A descriptor — a grammar slot, a left extent and a right extent — then uniquely identifies the application of a continuation function to a particular right extent. Thus, preventing the repeated application of the same continuation function to the same right extent is similar to preventing the repeated execution of a descriptor. As noted before, in Section 4.3 we give combinator definitions inspired by the strong similarity between Johnson's continuation management and that of FUN-GLL.

3.3.4 Comparison

Left-recursive parameterised combinators In §3.2.3, several complications are discussed regarding the use of $inject_R$ to define recursive and parameterised combinators. The same complications apply here, with the difference that only left-recursive definitions re-

[13] See also footnote 11.
[14] In their formulation, a rule is a collection of alternates.

quire the use of *inject* to prevent nontermination. (Although it is beneficial to use *inject* wherever memoisation makes a difference.) It is therefore not straightforward to define a left-recursive variant of *sepBy* or *scales*, introduced in §3.1.4 and §3.2.3 respectively.

In Johnson's SCHEME implementation, a combinator *memo* is defined which evaluates to a memoised closure of the combinator expression to which it is applied. A memo-table is created whenever an occurrence of *memo* is evaluated, and a reference to that table is stored within the environment of the closure. It is therefore important to take evaluation order into account. For example, when defining a left-recursive variant of *sepBy* it must be ensured that a memo-table is created after *sepBy* has been given its arguments and that the recursive call is replaced by the memoised closure. The following SCHEME fragment contains a working left-recursive definition of *sepBy*.

```
(define (sepBy P S)
   (letrec ((REC (memo (lambda (I) ((alt P (seq REC (seq S P))) I)))))
           REC))
```

Where we might expect a recursive call, we find `REC` instead, a memoised closure embedding all of the 'behaviour' of `sepBy` (the closure waits for input sentence `I` to be given). The following naive definition results in infinitely many applications of *sepBy* and thus infinitely many applications of *memo*.

```
(define (sepBy P Q) (memo (alt P (seq (sepBy P Q) (seq Q P)))))
```

We cannot define a left-recursive variant of *scales* in this fashion. This is because the recursive call of scales requires a modified argument, i.e. the recursive call in the definition of $scales(p)$ is $scales(parens(p))$.

Combinator laws We expect that it is possible to prove the laws of Figure 3.4 for the combinators of this section, with the exception of (right-absorption-seq) because, as a second operand of *seq*, *fails* will not remove any right extents or continuation functions added to the memo-table as part of the execution of the first operand. Proving distributivity, commutativity and idempotency of *alt* is difficult at best, since such proofs depend on commutativity and idempotency of $\S$, which in turn requires showing that certain invariants over the memo-table are maintained. An example invariant is: applying a parse function ψ a second time does not grow the memo-table and will yield the same set of right extents

as the original application (follows from the soundness of memoisation). When defining additional elementary combinators, one needs to prove that the invariants still hold in order to uphold the laws, making the library significantly harder to maintain.

Additional elementary combinators Although possible, it is more difficult to define additional elementary combinators compared to §3.1.1. Firstly, because, as noted above, in order to prove certain combinator laws, it is necessary that any additional elementary combinators maintain certain memo-table invariants. Secondly, it may be necessary to define new continuation functions, as shown by the definition of *peek* below.

$$\mathit{peek}(p)(c_0)(I,i) = p(c_1)(I,i) \quad \textbf{where } c_1(r) = c_0(i)$$

The continuation c_1 is necessary to 'reset' the current index, since *peek* is intended not to consume any input. This definition is not ideal, as c_1 may be applied multiple times. This happens if p recognises multiple subsentences of the input sentence, or it recognises a subsentence in more than one way, resulting in unnecessary applications of c_0. Rather, we prefer to use the initial continuation $\overline{c}$ to explicitly test whether p matches any subsentence.

$$\mathit{peek}(p)(c_0)(I,i)(M) = \begin{cases} c_0(i)(M') & \textbf{if } \langle M', R\rangle = p(\overline{c})(I,i)(M), R \neq \emptyset \\ \langle M, \emptyset\rangle & \text{otherwise} \end{cases}$$

The usage of $\overline{c}$ is similar to that in the definition of *recognise*. Note that the current memo-table M is used to evaluate p and that the resulting memo-table M' is used in the application of c_0, possible speeding up the evaluation of 'future' occurrences of p.

3.3.5 Discussion

This chapter revolves around a collection of elementary combinators of which variations are found in many existing implementations of combinator libraries, and introduces a combinator *inject* for making sharing observable. Throughout this chapter, several definitions of these combinators are discussed and compared.

Without observable sharing, the core combinators are already very useful. Parsers, and combinator expressions generally, can be written easily as the composition of existing expressions. Moreover, the collection of core combinators is relatively easy to extend. So-called derived combinators can be defined to capture common patterns, simply by applying exist-

ing combinators. It is also possible to define additional elementary combinators, like *pred* and *not*, although this requires a detailed understanding of the core definitions. Through library extensions, we have a method for defining reusable and executable components for syntax specification, the primary goal of this part of the thesis.

Most implementations of combinator libraries employ a simple recursive descent parsing algorithm, thus inheriting the well-known drawbacks of recursive descent. We have explained the work of several authors to overcome these drawbacks. By giving different implementations to *inject*, we have explained Ridge's grammar generating combinators and Johnson's memoising combinators. These implementation have limitations of their own. For example, they make it harder to define new elementary combinators, since the underlying algorithms are more complex. Ridge's grammar-generating combinators generate binarised grammars. In the next chapters we show that grammar binarisation can be avoided and in Section 13.1 we demonstrate the negative effects of binarisation on the effectiveness of FUN-GLL. Johnson's combinators are for recognition only. Although not shown here, it is possible to extend the underlying algorithm to produced extended packed node sets. This involves further complicating the underlying algorithm, since grammar information is required to compute the slots of the extended packed nodes.

Chapter 4

Explicit BNF Combinator Parsing

The previous chapter presents a core collection of parser combinators commonly found in libraries for functional programming languages like ML and Haskell. An essential feature of these combinators is that they are freely composed. This makes parser combinators easy to use and combinator libraries easy to extend. In the previous chapter we have seen the *inject* combinator that makes it possible to detect repeated evaluation by injecting additional informational into combinator expressions. The *inject* combinator was used to discuss the methods of Ridge and Johnson to generalising the parser combinators.

In this chapter we go a step further and introduce a novel collection of core combinators with stricter types so that combinator expressions have a richer, more informative structure. The combinator expressions represent BNF grammar specifications explicitly, and we therefore refer to the combinators of this chapter as BNF combinators. We show that the additional information of BNF combinator expressions can be exploited to generate grammars in a straightforward manner, avoiding binarisation, resulting in grammars with fewer nonterminals compared to their binarised counterparts. The additional information can also be used to compute grammar slots, descriptors and extended packed nodes. It is therefore possible to implement an algorithm like FUN-GLL directly, without the need for generating a grammar object.

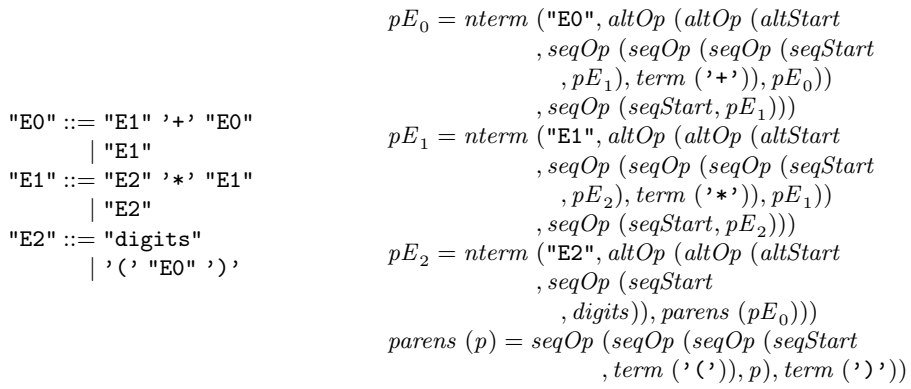


Figure 4.1: A combinator expression representing an arithmetic expression grammar.

4.1 Explicit BNF Combinators

Rather than one type of combinator expressions, the explicit BNF combinators introduced in this section form *symbol expressions*, *sequence expressions* and *choice expressions*. Symbol expressions represent symbols in BNF, a sequence expressions represents a sequence of symbols (an alternate), whereas a choice expression represents the choice between several alternates. The symbol expression $term(w)$ simply represents the terminal w. The sequence expression *seqStart* represents the empty sequence of symbols. Each application of *seqOp* adds an additional symbol (second argument) to an alternate (first argument). Combinator *seqOp* therefore relates to juxtaposition in BNF rules. Similarly, a choice expression constructed by *altStart* represents the empty sequence of alternates. Each application of *altOp* adds an additional alternate (second argument) to a sequence of alternates (first argument). Combinator *altOp* therefore relates to the | operator in BNF rules. An application of *nterm* groups a sequence of alternates (second argument) under a single label (first argument). Combinator *nterm* therefore relates to the ::= operator in BNF rules. For example, the BNF rule *opta* ::= ϵ | `'a'` is represented by the following symbol expression:

$$nterm(\texttt{"opta"}, alt(alt(altStart, seqStart), seq(seqStart, term(\texttt{'a'}))))$$

Figure 4.1 shows the BNF description of a grammar and a BNF combinator expression

$$
\begin{aligned}
&\mathit{term}(x) = \langle \mathit{nt}(x), \mathit{gen} \rangle \\
&\quad \textbf{where } \mathit{gen}(C, P) = \langle C, \{\mathit{nt}(x) ::= x\} \cup P \rangle \\
&\mathit{nterm}(l, \langle \alpha_1 \ldots \alpha_m, p_2 \rangle) = \langle x, \mathit{gen} \rangle \\
&\quad \textbf{where } x = \mathit{show}(l) \\
&\quad \textbf{and} \qquad \mathit{gen}(C, P) = \begin{cases} \langle C, P \rangle & \textbf{if } l \in C \\ p_2(C \cup \{l\}, P \cup \{x ::= \alpha_i \mid 1 \leqslant i \leqslant m\}) & \text{otherwise} \end{cases} \\
&\mathit{seqStart} = \langle \epsilon, \mathit{gen}_{\mathit{id}} \rangle \\
&\mathit{seqOp}(\langle \alpha, p_2 \rangle, \langle s, q_2 \rangle) = \langle \alpha s, \mathit{gen}_{\mathit{comp}}(p_2, q_2) \rangle \\
&\mathit{altStart} = \langle \epsilon, \mathit{gen}_{\mathit{id}} \rangle \\
&\mathit{altOp}(\langle \alpha_1 \ldots \alpha_{m-1}, p_2 \rangle, \langle \alpha_m, q_2 \rangle) = \langle \alpha_1 \ldots \alpha_{m-1} \alpha_m, \mathit{gen}_{\mathit{comp}}(p_2, q_2) \rangle \\
&\mathit{gen}_{\mathit{id}}(C, P) = \langle C, P \rangle \\
&\mathit{gen}_{\mathit{comp}}(g_1, g_2)(C, P) = (g_2 \circ g_1)(C, P)
\end{aligned}
$$

Figure 4.2: Generating a grammar from explicit BNF combinators.

that represents it. (Combinator *digits* is defined similarly as in §3.1.4.) The combinator expression is not very legible and cumbersome to write. In §5.3.3, a Haskell implementation is discussed with which combinator expressions are written more comfortably using infix operators. Moreover, the implementation uses coercions to automatically insert occurrences of *seqStart* and *altStart* where necessary.

4.2 Generating a Grammar

This section shows how the combinators presented in the previous section can be defined in order to generate the grammars explicitly represented by combinator expressions. The definitions are explained in direct comparison with the definitions of the conventional combinators given in §3.2.2 on page 50. Figure 4.2 gives the definitions of *term*, *nterm*, *seqOp*, *seqStart*, *altOp* and *altStart*, and relies on the functions *nt* and *show*, given in §3.2.2.

The first component of each type of combinator expression is the grammar fragment it represents. As in §3.2.2, the second component of each type of combinator expression is a function mapping a set of labels and a set of productions to a (possibly extended) set of labels and a (possibly extended) set of productions. Combinator *nterm* injects labels

to detect recursion and prevent nontermination. Note that where in §3.2.2 applying *inject* is optional, using *nterm* is required to produce non-trivial symbols expressions. In §3.2.2, each combinator expression adds productions to the computed grammar. Here, only symbol expressions add productions to the produced grammar. Combinator *nterm* adds one production for each alternate represented by its choice expression argument, but only if its label has not been encountered before. Combinator *term* is defined as $term_R$ in §3.2.2. Operators *seqOp* and *altOp* behave identically: both add an element (the first component of the second operand) to the end of a sequence (the first component of the first operand) whilst composing the second components of the operands in the same order. However, the operators work on combinator expressions of different types. Operator *seqOp* expects a sequence expression and a symbol expression whereas *altOp* expects a choice expression and a sequence expression.

We create a wrapper function, *grammarOf*, that given a label l and a symbol expression yields a grammar:

$$\begin{aligned} &grammarOf(l, \langle n, gen\rangle) = \langle show(l), P\rangle \\ &\quad \textbf{where } \langle C, P\rangle = gen(\emptyset, \{show(l) ::= n\}) \end{aligned}$$

The grammar is augmented with a start symbol based on the given label l, which is assumed not to be used in any application of *nterm* occurring in the given symbol expression. The definitions of Figure 4.2 are such that $grammarOf(\texttt{"Z"}, pE_0)$ evaluates to the grammar[1] shown on the left-hand side of Figure 4.1 augmented with the production `"Z"` ::= `"E0"`.

4.2.1 An Unrestricted Interface

This section shows that we can implement combinators *inject*, *seq* and *alt* in terms of *nterm*, *seqOp*, *seqStart*, *altOp* and *altStart*. We define *inject*, *seq* and *alt* in Figure 4.3 as operators on symbol expressions. (Note that in §3.2.2 all combinator expressions are essentially what we refer to as symbol expressions here.) This is achieved by turning a symbol expression representing symbol s into a sequence expression representing the singleton sequence s (using *singleS*), or into a choice expression representing a single alternate which contains just the

[1] As in Chapter 3, we use strings as labels and characters for terminals, thus *show* is the identity functions on strings and *nt* is a function from characters to strings.

$$
\begin{aligned}
\mathit{inject}(l, p) &= \mathit{nterm}(l, \mathit{singleA}(\mathit{singleS}(p))) \\
\mathit{seq}(p, q) &= \mathit{nterm}(\mathit{fst}(p) \times_{\#} \mathit{fst}(q), \mathit{singleA}(\mathit{seqOp}(\mathit{singleS}(p), q))) \\
\mathit{alt}(p, q) &= \mathit{nterm}(\mathit{fst}(p) +_{\#} \mathit{fst}(q), \mathit{altOp}(\mathit{singleA}(\mathit{singleS}(p)), \mathit{singleS}(q))) \\
\mathit{singleA}(s) &= \mathit{alt}(\mathit{altStart}, s) \\
\mathit{singleS}(x) &= \mathit{seq}(\mathit{seqStart}, x)
\end{aligned}
$$

Figure 4.3: Combinators *inject*, *seq* and *alt* defined in terms of *nterm*, *seqOp*, *seqStart*, *altOp* and *altStart*.

symbol s (using *singleA* and *singleS*). In order to generate nonterminals for occurrences of *seq* and *alt*, we use the operators $\times_{\#}$ and $+_{\#}$ introduced in §3.2.2.

We conjecture that combinator expressions formed by applications of *term*, *inject*, *seq* and *alt* generate the same grammar if they were formed by applications of term_R, inject_R, seq_R and alt_R instead. A combinator library based on the explicit BNF combinators can thus offer its users a choice between the somewhat restrictive interface provided by *nterm*, *seqOp*, *seqStart*, *altOp* and *altStart* and the unrestricted interface provided by *inject*, *seq* and *alt*, without implementing the underlying algorithms twice. In §5.3.3 we show that the restrictive interface can be made more user-friendly in a Haskell implementation, resulting in a flexible interface with automatic conversions between symbol expressions. In Section 13.1 we demonstrate the significant negative effects of grammar binarisation on the running times of FUN-GLL.

4.3 Direct FUN-GLL Parsing

The additional structure of combinator expressions formed by application of the explicit BNF combinators makes it possible to associate grammar slots with subexpressions. A grammar slot can then be seen to contain information about the context of the subexpression with which it is associated. This information can be used for several purposes. As shown in this section, the grammar slots can be used to form extended packed nodes, and thus to implement parsing algorithms. It is possible to define the combinators *term*, *nterm*, *seqOp*, *seqStart*, *altOp* and *altStart* in the style of Johnson's continuation-passing combinators

(§3.3.2) and to extend the algorithm to output extended packed nodes. Instead, this section defines the combinators based on the FUN-GLL algorithm of §2.4.3, thus using a call-graph ($\mathcal{G}$) and a pop-set ($\mathcal{P}$). The resulting combinator definitions are very similar to the definitions of §3.3.2, and we explain the definitions of this section as a modification. The functions *appToR*, *appToC*, and *contFor* will be redefined (they will have different types but similar behaviour and intent). With respect to FUN-GLL, the call-graph used here is slightly modified. The modified call-graph maps commencements to a descriptor without index (referred to as a continuation in §2.4.3), but now paired with a continuation function — in the sense of §3.3.2 — which 'does the remaining' work when applied. Continuations uniquely identify continuation functions. In §3.3.2, a continuation function c is a higher-order function that given an index r (right extent) returns a parse function. Here, a continuation function is also given the continuation that identifies it (thus a continuation function receives a grammar slot, a left extent and a right extent). This additional information is needed to construct extended packed nodes and to use descriptor set ($\mathcal{U}$) to avoid any repeated work resulting from applying continuation functions (see also §3.3.3) as shown by the definitions of *continue* and *seqOp* that follow.

State In §3.3.2, parse functions ψ are given a memo-table and return a possibly extended memo-table and a set of right extents. The right extents are the final outcome of the algorithm. Here, we are interested in computing a set of extended packed nodes $\mathcal{E}$. The memo-table is replaced by $\mathcal{U}$ (descriptor set), relation $\mathcal{G}$ (call-graph) and relation $\mathcal{P}$ (pop-set). A parse function is then an endofunction over quadruples of the form $\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle$ referred to as a state. The $\fatsemi$ operator is inverted function composition, e.g. $\psi_1 \fatsemi \psi_2 = \psi_2 \circ \psi_1$. Operator $\fatsemi$ is generalised so that $\fatsemi\{\psi_1, \ldots, \psi_n\}$ is defined as $\psi_0 \fatsemi \psi_1 \fatsemi \ldots \fatsemi \psi_n$ with ψ_0 the identity function over states.

We introduce *getP* for obtaining the right extents for a particular commencement $\langle s, i \rangle$ from the pop-set in a given state. Similarly, *getG* obtains the continuations and continuation function pairs for a particular commencement. The definitions of these functions are given in Figure 4.4. Functions for extending the call-graph and pop-set are also given in Figure 4.4.

$$getP(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle)(X, k) = \{r \mid \langle \langle X, k \rangle, r \rangle \in \mathcal{P}\}$$
$$getG(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle)(Y, k) = \{\langle X ::= \alpha Y \cdot \beta, l, c \rangle \mid \langle \langle Y, k \rangle, \langle X ::= \alpha Y \cdot \beta, l, c \rangle \rangle \in \mathcal{G}\}$$

$$addP(X, k, r)(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle) = \langle \mathcal{U}, \mathcal{G}, \mathcal{P} \cup \{\langle \langle X, k \rangle, r \rangle\}, \mathcal{E} \rangle$$
$$addG(Y, k, g, l, c)(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle) = \langle \mathcal{U}, \mathcal{G} \cup \{\langle \langle Y, k \rangle, \langle g, l, c \rangle \rangle\}, \mathcal{P}, \mathcal{E} \rangle$$

Figure 4.4: Accessor and modifier functions for parse function state.

Continuing In Johnson's algorithm, there are three types of continuation functions. There is the initial continuation function, which simply returns the right extent given to it as a singleton set. There are the continuation functions constructed as part of the definition of *seq* and the continuation functions constructed as part of the definition of *inject* (by applying *contFor*). To avoid repeated work, we associate a continuation $(X ::= \alpha \cdot \beta, l)$ with every continuation function c of the second kind. When c is applied to right extent r, we can check for the presence of the descriptor $(X ::= \alpha \cdot \beta, l, r)$ to determine whether c has been applied to r before. Doing this only for continuations created as part of the definition of *seq* (here *seqOp*) is sufficient as the continuations created as part of the definition of *inject* (here *nterm*) immediately apply a continuation of the second or first kind.

The function *continue* is given a continuation, a continuation function, a pivot and a right extent and applies the continuation function to the right extent, but only if it is the first time.

$$\begin{aligned}
&continue(X ::= \alpha \cdot \beta, l, k, r, c)(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle) = \\
&\quad \begin{cases} \langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle & \textbf{if } (X ::= \alpha \cdot \beta, l, r) \in \mathcal{U} \\ c(X ::= \alpha \cdot \beta, l, r)(\langle \mathcal{U} \cup \mathcal{U}_1, \mathcal{G}, \mathcal{P}, \mathcal{E} \cup \mathcal{E}_1 \rangle) & \text{otherwise} \end{cases} \\
&\quad \textbf{where } \mathcal{U}_1 = \{(X ::= \alpha \cdot \beta, l, r)\} \\
&\quad \textbf{and} \qquad \mathcal{E}_1 = \begin{cases} \{(X ::= \alpha \cdot \beta, l, k, r)\} & \textbf{if } \alpha \neq \epsilon \vee \beta = \epsilon \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Function *continue* also adds an extended packed node of the form $(X ::= \alpha \cdot \beta, l, k, r)$ but does so only if $\alpha \neq \epsilon$, or if $\alpha = \epsilon = \beta$, implying $l = k = r$.

The core combinators Function *continue* is applied whenever all the symbols represented by a sequence expression have successfully matched some part of the input sentence. An occurrence of *seqStart* immediately applies *continue*, as it represents the empty sequence of symbols.

$$\mathit{seqStart}(c)(I, X, \beta, l) = \mathit{continue}(X ::= \cdot\beta, l, l, l, c)$$

A sequence expression receives the arguments (I, X, β, l) such that: X is the nonterminal symbol representing the closest occurrence of *nterm* 'above' this sequence expression and β represents the symbols occurring 'later' in the sequence. Left extent l is the index given to the closest occurrence of *nterm* 'above' this sequence expressions. The definition of *seqOp* below shows how this information is propagated. In this definition, q is the second component of a symbol expression and the definitions of *term* and *nterm*, for creating symbol expressions, are given later.

$$\begin{aligned}
&\mathit{seqOp}(p, \langle s, q\rangle)(c_0)(I, X, \beta, l) = p(c_1)(I, X, s\beta, l)\\
&\quad \textbf{where } c_1(X ::= \alpha \cdot s\beta, l, k) = q(c_2)(I, X, \alpha, \beta, l, k)\\
&\qquad\qquad \textbf{where } c_2(X ::= \alpha s \cdot \beta, l, r) = \mathit{continue}(X ::= \alpha s \cdot \beta, l, k, r, c_0)
\end{aligned}$$

When the first operand p of *seqOp* is applied, β is extended with the symbol s of the second operand. When continuation c_1 is applied to $(X ::= \alpha \cdot s\beta, l, k)$, this indicates that the subsentence $I_{l,k}$ has been recognised by p and that this involved the symbols α of the alternate $X ::= \alpha s\beta$. When c_2 is applied, this must be to $(X ::= \alpha s \cdot \beta, l, r)$, indicating that q has recognised $I_{k,r}$ and thus that $\mathit{seqOp}(p, \langle s, q\rangle)$ has recognised $I_{l,r}$. The dot is 'moved across' s, the symbol represented by the second operand, and the original continuation c_0 is applied via *continue* with pivot k.

Above, q is the second component of a symbol expression, and receives as arguments X, α and β — but not the symbol s it represents — and the pivot k as current index. If the symbol expression recognises $I_{k,r}$ for some r, it applies the given continuation, extending α with s. This is shown by the definitions of *term* and *nterm* below.

$$\begin{aligned}
&\mathit{term}(t) = \langle t, f\rangle\\
&\quad \textbf{where } f(c)(I, X, \alpha, \beta, l, k)(\sigma) = \begin{cases} c(X, \alpha t, \beta, l, k+1)(\sigma) & \textbf{if } I_k = t\\ \sigma & \text{otherwise}\end{cases}
\end{aligned}$$

As in the case of *inject*, the definition of *nterm* involves checking whether its label has been used to match with the same current index before, for which FUN-GLL uses the pop-set. In the definition below, p is a choice expression and the definitions of *altOp* and *altStart*, for creating choice expressions, are given later.

$$
\begin{aligned}
&\mathit{nterm}(s,p) = \langle s, f\rangle\\
&\quad \textbf{where } f(c)(I,X,\alpha,\beta,l,k)(\sigma) =\\
&\qquad \begin{cases} \mathit{appToR}(s,k,\langle X ::= \alpha s\cdot\beta,l,c\rangle,R)(\sigma') & \textbf{if } R\neq\emptyset\\ p(\mathit{contFor}(s,k))(I,s,k)(\sigma') & \text{otherwise}\end{cases}\\
&\qquad \textbf{where } R = \mathit{getP}(\sigma)(s,k)\\
&\qquad \textbf{and}\quad\ \ \sigma' = \mathit{addG}(s,k,X ::= \alpha s\cdot\beta,l,c)(\sigma)
\end{aligned}
$$

Note the subtle difference between the condition in *nterm* above and *inject* in §3.3.2. Here, p may be applied a second time, if in the meantime no elements have been recorded in the pop-set. In this case, the second call will result in an application of *seqStart*, which will apply *continue* with the same arguments and terminate. This slight inefficiency has not been removed to retain the close correspondence with FUN-GLL. The third argument of *appToR* (which is redefined later) is a continuation and continuation function pair and its fourth argument is a set of right extents.

When the continuation $\mathit{contFor}(s,k)$ is applied to $(x ::= \alpha\cdot\beta,k',r)$ we know that $x = s$, that $\beta = \epsilon$ and that $k' = k$. Continuation $\mathit{contFor}(s,k)$ may be applied, multiple times, to different values of r and α, however.

$$
\begin{aligned}
&\mathit{contFor}(s,k)(s ::= \alpha\cdot\epsilon,k,r)(\sigma) = \mathit{appToC}(s,k,r,C)(\sigma')\\
&\quad \textbf{where } C = \mathit{getG}(\sigma)(s,k)\\
&\quad \textbf{and}\quad\ \ \sigma' = \mathit{addP}(s,k,r)(\sigma)
\end{aligned}
$$

When $\mathit{contFor}(s,k)$ is applied, this results in the application of all the continuation functions stored in the call-graph to the given right extent r. These applications may terminate directly when a duplicate descriptor is encountered.

The definitions of *appToR* and *appToC* are as follows:

$$
\begin{aligned}
\mathit{appToR}(s,k,\langle X ::= \alpha s\cdot\beta,l,c\rangle,R) &= {}^{\circ}_{9}\{c(X ::= \alpha s\cdot\beta,l,r) \mid r\in R\}\\
\mathit{appToC}(s,k,r,C) &= {}^{\circ}_{9}\{c(Y ::= \delta s\cdot\nu,l,r) \mid \langle Y ::= \delta s\cdot\nu,l,c\rangle\in C\}
\end{aligned}
$$

The second operand of *nterm* is a choice expression which receives a commencement as

argument, together with the input sentence. An occurrence of *altStart* corresponds to *fails*, meaning that it does not apply the given continuation and returns the state unchanged. The definition of *altOp*, given below, shows that the initial symbol sequence β given to a sequence expression is empty (ϵ).

$$
\begin{aligned}
&\mathit{altStart}(c)(I, X, l)(\sigma) = \sigma \\
&\mathit{altOp}(p, q)(c)(I, X, l) = p(c)(I, X, l) \fatsemi q(c)(I, X, \epsilon, l)
\end{aligned}
$$

The function *parse*, defined below, generates a parser — a function from an input sentence I to a set of extended packed nodes $\mathcal{E}$ — given a symbol expression $p = \langle s, f \rangle$. In order to define *parse*, it is necessary to apply f, the second component of p, which requires a symbol l, the empty state $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$, and the base continuation $\overline{c}(g, k, r)(\sigma) = \sigma$.

$$
\begin{aligned}
&\mathit{parse}(l, \langle s, f \rangle)(I) = \mathcal{E} \\
&\quad \textbf{where } \langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{E} \rangle = f(\overline{c})(I, l, \epsilon, \epsilon, 0, 0)(\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle) \\
&\quad \textbf{and} \quad \overline{c}(g, k, r)(\sigma) = \sigma
\end{aligned}
$$

The first argument of *parse* is a symbol l which is assumed to be unique in the sense that it does not occur as an argument to any occurrence of *nterm* within p. The symbol acts as the parent of s, the first component of p, in the grammar slots of the continuations and extended packed nodes that are created by the algorithm. Note that the set of extended packed nodes $\mathcal{E}$ is not used in the definitions of the combinators; its only purpose is to collect derivation information as output.

Comparison A definition of *inject*, *seq* and *alt* can be given based on the combinators of this section, if labels are strings, exactly as we have done for the grammar-generating variant in §4.2.1.

Compared to §3.3.2, the combinators defined here use a call-graph, a pop-set and a set of descriptors. The set of descriptors is used to avoid more duplicate work, achieving a similar effect to memoising all continuation functions as suggested by Izmaylova, Afroozeh and Van der Storm [Izmaylova et al., 2016], briefly discussed in §3.3.3. Because of the similarity with Johnson's combinators, we can make similar arguments to those of §3.3.4 regarding recursive parameterised combinators, adding additional elementary combinators and combinator laws. Continuation functions are more complicated, as they receive a descriptor as

argument, which is necessary for avoiding duplicate work and computing extended packed nodes. Therefore, defining additional elementary combinators is more challenging. On the other hand, extended packed nodes are only created when evaluating sequence expressions. This means that only user-defined alternatives to *seqStart* and *seqOp* need to take extended packed nodes into account.

4.3.1 Discussion

In this chapter we introduced a novel collection of core combinators with which BNF grammar descriptions are explicitly represented by combinator expressions. The explicit representation of BNF grammar descriptions is achieved by restricting the signatures of the core combinators, giving combinator expressions more structure. It is possible to extract grammars from combinator expressions and apply FUN-GLL to the extracted grammars. The combinators can also be defined as parser combinators directly, computing extended packed nodes. The parsing algorithms underlying both versions of the BNF combinators are fully general with respect to the represented grammars, and can be combined with a version of Ridge's semantic phase in which semantic actions are executed. The next chapter implements FUN-GLL, the grammar-generating version of the BNF combinators, and a variant of Ridge's semantic phase based on extended packed nodes, and shows how these implementations are combined to form a practical combinator library.

Chapter 5

Haskell: Reusable Components for Syntax Specification

References *The implementations in Sections 5.1 and 5.3 of this chapter have been published as [Van Binsbergen et al., 2018].*

This chapter presents a literate Haskell implementation of the FUN-GLL algorithm of Chapter 2 in Section 5.1 and of the grammar-generating BNF combinators of Chapter 4 in Section 5.3. The user-friendliness of the BNF combinators is enhanced using Haskell's type-class mechanism [Hall et al., 1994]. The resulting interface is flexible in the sense that the user has the opportunity to avoid — or take advantage of — nonterminal generation and that combinator expressions are freely composed.

Parsers written with conventional parser combinators typically perform semantic evaluation on the fly by associating semantic functions with parsers. We explain the integration of semantic functions by revisiting the conventional parser combinators of Chapter 3 in Section 5.2 and show that they implement the *Applicative* and *Alternative* interfaces. To integrate semantic functions into the BNF combinators, we implement a variant of Ridge's semantic phase driven by extended packed nodes.

5.1 FUN-GLL Implementation

This section discusses technical details of an implementation of the FUN-GLL algorithm of 2.4.3. The next subsection implements the relevant concepts from Chapter 2 and summarises these concepts from an operational perspective.

5.1.1 Grammars and Derivations

A grammar is implemented as a mapping from nonterminals to sets of right-hand sides (also called alternates), where a right-hand side is a list of symbols. A symbol is either a terminal (a value of some type t) or a nonterminal (a value of some type n):

```
type Grammar n t = M.Map n (S.Set (Rhs n t))
type Rhs n t     = [Symbol n t]
data Symbol n t  = Term t | Nt n
type Input t     = Array Int t
instance (Show a, Show b) ⇒ Show (Symbol a b) where
  show (Term t) = show t
  show (Nt s) = show s
```

(Maps are imported from *Data.Map* under the qualified name M and sets are imported from *Data.Set* under the qualified name S.) Parser input is represented as an array of terminals. In our examples we use *String* for nonterminals and *Char* for terminals.

A nonterminal in a grammar derives a sequences of terminal symbols (sentences). Sentence I is derived by nonterminal x if I can be obtained by choosing a right-hand side of x and repeatedly replacing nonterminals within it by one of their right-hand sides. The set of sentences that can be derived from a nonterminal x in a grammar *gram* is the language generated by x in *gram*. For example, nonterminal `"tuple"` generates the language {`"()"`, `"(a)"`, `"(a,a)"`, ...} in the grammar:

```
tupleRR =  M.fromListWith S.union
  [("tuple", S.singleton [Term '(', Nt "as", Term ')'])
  ,("as"   , S.singleton [])   -- empty right-hand side
  ,("as"   , S.singleton [Term 'a', Nt "more"])
  ,("more" , S.singleton [])   -- empty right-hand side
  ,("more" , S.singleton [Term ',', Term 'a', Nt "more"])]
```

A recognition procedure is an algorithm that, given a grammar, a nonterminal x, and a sentence I, determines whether I is in the language generated by x in the grammar.

A parsing procedure is a recognition procedure that provides proof that this is the case, typically in the form of a derivation tree. A recognition or parsing procedure is general if it terminates and gives correct results for all grammars. A procedure is complete if it is general and provides proof for all possible derivations of the input sentence. A grammar is ambiguous if one sentence has multiple derivations.

FUN-GLL construct sets of extended packed nodes (EPNs) containing sufficient information to construct a complete BSPPF (see §2.1.4).

```
type Slot n t   = (n, Rhs n t, Rhs n t)
type EPN n t    = (Slot n t, Int, Int, Int)
type EPNs n t ≡ S.Set (EPN n t)
```

We use ≡ to *suggest* a possible definition for EPNs. In actuality we use a more efficient definition based on Patricia trees [Okasaki and Gill, 1998].

A *grammar slot* is a triple (x, α, β) where $\alpha\beta$ is an alternate of nonterminal x. An EPN is a quadruple $((x, \alpha, \beta), l, k, r)$, with (x, α, β) a grammar slot and integers $l \leqslant k \leqslant r$.

The following operations are used to construct EPN sets:

```
emptyPNs    :: EPNs n t
singlePN    :: EPN n t    → EPNs n t
unionPNs    :: EPNs n t   → EPNs n t → EPNs n t
unionsPNs   :: [EPNs n t] → EPNs n t
unionsPNs = foldr unionPNs emptyPNs
fromListPNs :: [EPN n t]  → EPNs n t
fromListPNs = foldr (unionPNs ∘ singlePN) emptyPNs
```

We can use standard equality (Haskell's type-class *Eq*) to match the terminal symbols of an input sentence to the terminals in a grammar. However, it may be useful to use a different notion of equality. A particular use case is that terminals are produced by lexers such that the terminals embed their lexemes. These kinds of terminals can be implemented as shown by the *Token* type defined below:

```
data Token = Char     Char
           | Keyword  String
           | IntLit   (Maybe Int)
           | FloatLit (Maybe Double)
           | BoolLit  (Maybe Bool)
           | StringLit (Maybe String)
           | CharLit  (Maybe Char)
           | IDLit    (Maybe String)
           | AltIDLit (Maybe String)
           | Token String (Maybe String)
```

The second argument of the *Token* constructor, and the arguments of constructors for the different kind of literals, are optional lexemes. The intention is that, for example, *IDLit Nothing* appears in a grammar, whereas *IDLit* (*Just* `"blueberry"`) is produced by a lexer.

We introduce a type-class containing those types whose values can be 'matched'. The default definition of the method *matches* is equality.

```
class (Ord a, Eq a) ⇒ IsTerminal a where
  matches :: a → a → Bool
  matches = (≡)   -- defaults to equality
```

The case studies of this thesis use the *Token* type as the instantiation for terminals in grammars and input sentences. The *Token* constructor can be used to extend the *Token* type arbitrarily, using *Strings* to identity additional forms. To extend the *Token* type in a way that gives more type-safety, the *Token* type can be wrapped in a type that provides additional token forms. Such a wrapper type is said to 'subsume' the *Token* type if there is a projection and injection function defined between the two types:

```
class SubsumesToken a where
  upcast   :: Token → a          -- inject a Token into the wrapper type
  downcast :: a → Maybe Token    -- project a if a is indeed a wrapped Token
```

5.1.2 Summary of FUN-GLL

Nontermination due to left-recursion, and other kinds of 'repeated work', are prevented by introducing descriptors (Section 2.3). The complexity is potentially $O(n^3)$ in both space and runtime, depending on the implementation of the data structures.

A descriptor is a triple containing a slot, a left extent and another index:

```
type Descr n t = (Slot n t, Int, Int)
```

A worklist $\mathcal{R}$ contains descriptors that require processing; its elements are processed one by one by calling a function called *process*. A set $\mathcal{U}$ of descriptors remembers the descriptors added to $\mathcal{R}$, and is used to ensure that no descriptor is added to the worklist a second time. Preventing repeated processing avoids nontermination due to left-recursion, but may result in the omission of derivation information.

Consider the situation in which the descriptor $((x, \alpha, s : \beta), l, k)$ has been processed, with s a nonterminal, having resulted in further descriptors $((x, \alpha +\!\!+ [s], \beta), l, r_i)$, for all r_i in some set R. If the next processed descriptor is of the form $((y, \delta, s : \nu), l', k)$, then no further descriptors are added to $\mathcal{R}$, as all descriptors of the form $((s, [\,], \mu), k, k)$ have already been encountered. However, since s derives the subsentences ranging from k to $r_i - 1$, with $r_i \in R$, we should still add the descriptors $((y, \delta +\!\!+ [s], \nu), l', r_i)$ and EPNs $((y, \delta +\!\!+ [s], \nu), l', k, r_i)$. To avoid missing these descriptors and EPNs, the binary relation $\mathcal{P}$ between pairs of commencements and right extents is used, where a commencement is a pair of a nonterminal and a left extent. In the example situation, the set R is embedded in $\mathcal{P}$ as specified by the equation $R = \{r \mid ((s, k), r) \in \mathcal{P}\}$.

This is not sufficient; some of the descriptors of the form $((s, [\,], \mu), k, k)$, or descriptors that follow from these, may not have been processed yet. This means that there may be right extents R', with $R' \cap R = \emptyset$, for which it holds that s derives the subsentences ranging from k to $r_j - 1$, with $r_j \in R'$. When the right extents in R' are 'discovered', it is necessary to add the descriptors $((y, \delta +\!\!+ [s], \nu), l', r_j)$ and $((x, \alpha +\!\!+ [s], \beta), l, r_j)$ as well as the EPNs $((y, \delta +\!\!+ [s], \nu), l', k, r_j)$ and $((x, \alpha +\!\!+ [s], \beta), l, k, r_j)$. The binary relation $\mathcal{G}$ between commencements and *continuations* is used for this purpose, where a continuation is a pair of a slot and a left extent (i.e. a descriptor missing an index).

```
type Comm n t = (n, Int)
type Cont n t = (Slot n t, Int)
```

The FUN-GLL algorithm is summarised as follows: While there are descriptors in the worklist, arbitrarily select the next descriptor $((x, \alpha, \beta), l, k)$ to be processed, and

- If $\beta = w : \beta'$ and w is terminal, **match** the terminal at position k in the input sentence with w. Only if the match is successful, add the descriptor $((x, \alpha +\!\!+ [w], \beta'), l, k + 1)$ to the worklist and add EPN $((x, \alpha +\!\!+ [w], \beta'), l, k, k + 1)$ to the EPN set.

- If w is nonterminal, find $R = \{r \mid ((w, k), r) \in \mathcal{P}\}$ and extend $\mathcal{G}$ with $((w, k), ((x, \alpha +\!\!+ [w], \beta'), l))$. If R is empty, **descend** w by adding $((w, [\,], \delta), k, k)$, for all alternates δ of w, to the worklist (if not in $\mathcal{U}$). If R is not empty, **skip** w by adding the descriptors $((x, \alpha +\!\!+ [w], \beta'), l, r_i)$, for all $r_i \in R$ (if not in $\mathcal{U}$) and adding the EPNs $((x, \alpha +\!\!+ [w], \beta'), l, k, r_i)$ to the EPN set.

```
type RList n t ≡ S.Set (Descr n t)
popRList       :: RList n t → (Descr n t, RList n t)
emptyRList     :: RList n t
singletonRList :: Descr n t   → RList n t
unionRList     :: RList n t → RList n t → RList n t
fromListRList  :: [Descr n t] → USet n t → RList n t
fromListRList ds U = foldr op emptyRList ds
  where op d R | hasDescr d U = R
               | otherwise = unionRList (singletonRList d) R
type USet n t ≡ S.Set (Descr n t)
emptyUSet      :: USet n t
addDescr       :: Descr n t → USet n t → USet n t
hasDescr       :: Descr n t → USet n t → Bool
type GRel n t ≡ S.Set (Comm n t, Cont n t)
emptyG         :: GRel n t
addCont        :: Comm n t → Cont n t → GRel n t → GRel n t
conts          :: Comm n t → GRel n t → [Cont n t]
type PRel n t ≡ S.Set (Comm n t, Int)
emptyP     :: PRel n t
addExtent  :: Comm n t → Int → PRel n t → PRel n t
extents    :: Comm n t → PRel n t → [Int]
```

Figure 5.1: Types of FUN-GLL data structures and operations.

- If $\beta = [\,]$, extend $\mathcal{P}$ with $((x, l), k)$, and **ascend** x by finding all continuations $K = \{(slot, l') \mid ((x, l), (slot, l')) \in \mathcal{G}\}$, adding $(slot, l', k)$ to the worklist for all $(slot, l') \in K$ (if not in $\mathcal{U}$), adding EPNs $(slot, l', l, k)$ to the EPN set and, if $\alpha = \beta = [\,]$, add EPN $((x, [\,], [\,]), k, k, k)$ as well.

5.1.3 Data Structures

The types of the data structures and their operations are given in Figure 5.1. The efficiency and worst-case complexity of FUN-GLL is strongly influenced by the implementation of the data structures. We described the essential data structures — $\mathcal{R}$, $\mathcal{U}$, $\mathcal{G}$ and $\mathcal{P}$ — as sets, but a direct implementation as Haskell sets (from *Data.Set*) is inefficient. For example, we need to be able to determine quickly whether a descriptor has already been encountered by inspecting $\mathcal{U}$. This operation needs to be performed in constant time for FUN-GLL to have a worst-case complexity of $O(n^3)$ [Johnstone and Scott, 2011]. We do not discuss actual

implementations of these data structures and operations as we focus on the logic of the algorithm instead. We use $\equiv$ in the type definitions to suggest a simple implementation, as we did for *EPNs*.

The operation *popRList* arbitrarily removes an element from worklist $\mathcal{R} :: RList$. The worklist is constructed by applying *fromListRList* to a list of descriptors, guaranteeing that the resulting worklist contains no elements already in the given $\mathcal{U} :: USet$. The relation $\mathcal{G} :: GRel$, between commencements and continuations, is extended by applying *addCont*. Operation *conts* returns all the continuations paired with a given commencement in $\mathcal{G}$. The relation $\mathcal{P} :: PRel$, between commencements and right extents, is extended by applying *addExtent*. Similar to *conts*, *extents* returns all the right extents paired with a particular commencement in $\mathcal{P}$.

5.1.4 The Algorithm

Function *fungll* implements FUN-GLL. It is given a grammar, a nonterminal, and an input sentence and returns the processed set of descriptors $\mathcal{U}$ and the set of discovered EPNs, recursively applying function *loop* to $\mathcal{R}$ to process one descriptor at a time.

$$\begin{array}{l} \mathit{fungll}\ \mathit{gram}\ x\ \mathit{inp} = \mathit{loop}\ \mathit{gram}\ \mathit{inp}\ \mathcal{R}\ \mathit{emptyUSet}\ \mathit{emptyG}\ \mathit{emptyP}\ \mathit{emptyPNs} \\ \textbf{where}\ \mathcal{R} = \mathit{fromListRList}\ (\mathit{descend}\ \mathit{gram}\ x\ 0)\ \mathit{emptyUSet} \end{array}$$

The initial $\mathcal{R}$ contains the descriptors resulting from 'descending' the given symbol x with left extent 0. Descending a nonterminal x with index k requires the descriptors $((x, [\,], \beta), k, k)$ to be processed, for each alternate β of x.

$$\begin{array}{l} \mathit{descend}\ \mathit{gram}\ x\ k = [((x, [\,], \beta), k, k) \mid \beta \leftarrow \mathit{alts}] \\ \quad \textbf{where}\ \mathit{alts} = \mathit{maybe}\ [\,]\ S.\mathit{toList}\ (M.\mathit{lookup}\ x\ \mathit{gram}) \end{array}$$

Function *loop*, defined below, recurses over $\mathcal{R}$, removing a descriptor d (using *popRList*) for processing, until $\mathcal{R}$ is empty. The order in which descriptors are selected is irrelevant to the correctness (and worst-case complexity) of the algorithm, but might influence efficiency. An analysis of the influence of the order on efficiency is found in [Scott and Johnstone, 2016].

Processing a descriptor may result in descriptors, which are added to $\mathcal{R}$ if not in $\mathcal{U}$, as well as some EPNs. Processing a descriptor may also result in an extension to $\mathcal{G}$ or $\mathcal{P}$. Thus, *process* returns (a list of) descriptors, a (set of) EPNs, an optional commencement

and continuation pair, and an optional commencement and right extent pair. The list of descriptors returned by *process* is converted into an *RList* using *fromListRList*.

```
loop gram inp R U G P ns
  | null R     = (U, ns)   -- base case: R is empty
  | otherwise = loop gram inp rset'' uset' grel' prel' (unionPNs ns ns')
  where ((rlist, ns'), mcont, mpop) = process gram inp d G P
        (d, rset') = popRList R
        rset''     = unionRList rset' (fromListRList rlist uset')
        uset'      = addDescr d U
        grel'      | Just (k, v) <- mcont = addCont k v G
                   | otherwise            = G
        prel'      | Just (k, v) <- mpop  = addExtent k v P
                   | otherwise            = P
```

There are two cases to distinguish when a descriptor $((x, \alpha, \beta), l, k)$ is processed: $\beta = w : \beta'$ and $\beta = [\,]$.

```
process gram inp ((x, α, []), l, k) G P =
  ((rlist, unionPNs ns ns'), Nothing, Just ((x, l), k))
  where (rlist, ns) = ascend l K k
        ns' | α ≡ []     = singlePN ((x, [], []), l, k, k)
            | otherwise = emptyPNs
        K = [(slot, l') | (slot, l') <- conts (x, l) G]
```

The code above implements the latter case, in which it is discovered that x derives the subsentence ranging from l to $k-1$ and thus that k is a right extent for (x, l). This is 'remembered' by returning the commencement and right extent pair $((x, l), k)$ to extend $\mathcal{P}$. All continuations K, 'waiting' for the discovery of additional right extents such as k, are obtained by applying *conts* to (x, l) and $\mathcal{G}$. The descriptors and EPNs obtained by combining the continuations in K with l and k are returned by *ascend* (given later). The former case, $\beta = w : \beta'$ is implemented by the code below.

```
process gram inp ((x, α, w : β'), l, k) G P = case w of
  Term t ->  (match inp ((x, α, w : β'), l, k), Nothing, Nothing)
  Nt y
    | R ≡ [] -> ((descend gram y k, emptyPNs), Just cc, Nothing)
    | R ≢ [] -> (skip k ((x, α ++ [w], β'), l) R, Just cc, Nothing)
      where R = extents (y, k) P
            cc = ((y, k), ((x, α ++ [w], β'), l))
```

When w is a nonterminal symbol, the commencement and continuation pair $((w, k), ((x, \alpha +\!\!+ [w], \beta'), l))$ is returned for extending $\mathcal{G}$. Operation *extents* is used to find any right extents $r \in R$, providing the information that w derives the subsentence ranging from l to $r-1$.

If $R \equiv [\,]$, w is descended with left extent k (potentially for a second time). Otherwise, if $R \not\equiv [\,]$, function *skip* computes the descriptors and EPNs that follow from the earlier discovery that w derives the subsentence ranging from k to $r-1$. Function *skip* combines a single continuation with perhaps many right extents, whereas *ascend* combines a single right extent with potentially many continuations.

```
skip k d R      = nmatch k [d] R
ascend k K r    = nmatch k K [r]
nmatch k K R = (rlist, fromListPNs elist)
  where rlist = [(slot, l, r)    | (slot, l) ← K, r ← R]
        elist = [(slot, l, k, r) | (slot, l) ← K, r ← R]
```

If $\beta = w : \beta'$, and w is terminal, then *matches* from type-class *IsTerminal* is applied to check whether w matches the terminal at position k in the input sentence. If so, the descriptor $((x, \alpha +\!\!+ [w], \beta'), l, k+1)$ and the EPN $((x, \alpha +\!\!+ [w], \beta'), l, k, k+1)$ are returned.

```
match inp (slot@(x, α, Term t : β), l, k)
  | lb ⩽ k, k ⩽ ub, matches (inp ! k) t =
    ([((x, α ++ [Term t], β), l, k + 1)], singlePN ((x, α ++ [Term t], β), l, k, k + 1))
  | otherwise = ([], emptyPNs)
  where (lb, ub) = bounds inp
```

We demonstrate the generality of *fungll* by applying it to one of Ridge's example grammars [Ridge, 2014]:

```
tripleE = M.fromListWith S.union
            [("E", S.singleton [Nt "E", Nt "E", Nt "E"])
            ,("E", S.singleton [Term '1'])
            ,("E", S.singleton [])]
```

Grammar *tripleE* is left-recursive and cyclic[1], admitting infinitely many derivations of the sentences in the language it generates. Applied to *tripleE*, nonterminal `"E"`, and the input sentence `"1"`, *fungll* returns the following EPNs:

```
> printPNs (snd (fungll tripleE "E" (listArray (0,0) "1")))
1 : (("E", [], []), 0, 0, 0)
2 : (("E", ["E"], ["E", "E"]), 0, 0, 0)
3 : (("E", ["E", "E"], ["E"]), 0, 0, 0)
4 : (("E", ["E", "E", "E"], []), 0, 0, 0)
5 : (("E", ['1'], []), 0, 0, 1)
6 : (("E", ["E"], ["E", "E"]), 0, 0, 1)
7 : (("E", ["E", "E"], ["E"]), 0, 0, 1)
```

[1] A cyclic nonterminal can derive itself via one or more steps.

```
8  : (("E", ["E", "E"], ["E"]), 0, 1, 1)
9  : (("E", ["E", "E", "E"], []), 0, 0, 1)
10 : (("E", ["E", "E", "E"], []), 0, 1, 1)
11 : (("E", [], []), 1, 1, 1)
12 : (("E", ["E"], ["E", "E"]), 1, 1, 1)
13 : (("E", ["E", "E"], ["E"]), 1, 1, 1)
14 : (("E", ["E", "E", "E"], []), 1, 1, 1)
```

5.2 Backtracking Recursive Descent Evaluators

In this section we revisit the definition of the conventional parser combinators given in §3.1.1, and give an implementation based on the list-of-successes method from [Wadler, 1985], showing especially how semantic functions are integrated.

5.2.1 Semantic Values

The type *Parser t a* captures parse functions matching terminals of type t and returning semantic values of type[2] a.

$$\textbf{newtype}\ \mathit{Parser}\ t\ a = \mathit{Parser}\ (\mathit{Input}\ t \to \mathit{Int} \to [(\mathit{Int}, a)])$$

Values of type *Parser* correspond closely to the parse functions of §3.1.1 with the difference that right extents are paired with semantic values and that a multitude of results is collected in a list rather than a set. If p is a parse function and (r, a) is in $p\ inp\ l$, then p matches $inp_{l,r}$, the subsentence of inp ranging from l to $r-1$, and a is the semantic interpretation of $inp_{l,r}$. An example is provided by the *term* combinator.

$$\begin{aligned}
&\mathit{term} :: \mathit{IsTerminal}\ t \Rightarrow t \to \mathit{Parser}\ t\ t\\
&\mathit{term}\ t = \mathit{Parser}\ pf\\
&\quad \textbf{where}\ pf\ inp\ k \mid lb \leqslant k, k \leqslant ub, \mathit{matches}\ (inp\,!\,k)\ t = [(k+1, inp\,!\,k)]\\
&\qquad\qquad\quad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\quad = [\,]\\
&\qquad\quad \textbf{where}\ (lb, ub) = \mathit{bounds}\ inp
\end{aligned}$$

The semantic value is the matched terminal, taken from the input sentence rather than the grammar.

In what follows we use three methods for 'running' parse functions, e.g. recognition, evaluating, and counting.

[2] Given as a **newtype** rather than a type synonym to give type-class instances later.

```
evaluator :: Parser t a → [t] → [a]
evaluator (Parser pf) str = map snd $ filter complete $ pf inp 0
  where inp              = listArray (0, m − 1) str
        m                = length str
        complete (k, a)  = k ≡ m
counter :: Parser t a → [t] → Int
counter p = length ∘ evaluator p
recogniser :: Parser t a → [t] → Bool
recogniser p = (0 ≢) ∘ counter p
```

Function *evaluator p* is a function producing all interpretations of an input sentence[3] according to p, but only those interpretations that correspond to a complete match of the sentence. Function *counter p* produces a function that computes how many interpretations exist according to p. Function *recogniser p* is a function returning *True* if and only if the entire input sentence can be matched by p.

Curried semantic functions To explain how arbitrary semantic functions can be integrated in combinator expressions, we first show the signature of *seq*:

```
seq :: Parser t (a → b) → Parser t a → Parser t b
```

(Note that we only combine parsers that match the same type of terminal symbols.) An occurrence of *seq* constructs a parse function that produces semantic values of type b, given a parse function producing a semantic values of type $a \to b$, and a parse function producing semantic values of type a. The value of type b can only be obtained by applying the value of the first operand to the value of the second operand (since the type variables are universally quantified implicitly). The type b can be a function type, and thus we have a way of applying arbitrary curried functions[4] to parse results.

```
seq (Parser p) (Parser q) = Parser pf
  where pf inp l = [(r, f a) | (k, f) ← p inp l, (r, a) ← q inp k]
```

What remains is to show how arbitrary functions are introduced 'to the world of the parse functions'. In other words, how do we turn a value of type $a \to b$ into a value of type *Parser t* $(a \to b)$? The *succeeds* combinator is used for this purpose:

```
succeeds :: f → Parser t f
succeeds f = Parser pf
  where pf inp k = [(k, f)]
```

[3] We use input sentence ambiguously for values of type $[t]$ and *Input t*.

[4] We use "semantic functions" rather than "semantic actions" to emphasise the purity of the functions.

$$
\begin{aligned}
\mathit{alt}\ \mathit{fails}\ q &= q && \text{(left-identity-alt)}\\
\mathit{alt}\ p\ \mathit{fails} &= p && \text{(right-identity-alt)}\\
\mathit{alt}\ p\ (\mathit{alt}\ q\ r) &= \mathit{alt}\ (\mathit{alt}\ p\ q)\ r && \text{(associativity-alt)}\\
\\
\mathit{seq}\ (\mathit{succeeds}\ \mathit{id})\ q &= q && \text{(left-identity-seq)}\\
\mathit{seq}\ \mathit{fails}\ q &= \mathit{fails} && \text{(left-absorption)}\\
\mathit{seq}\ p\ \mathit{fails} &= \mathit{fails} && \text{(right-absorption)}\\
\mathit{seq}\ (\mathit{seq}\ (\mathit{seq}\ (\mathit{succeeds}\ (\circ))\ p)\ q)\ r &= \mathit{seq}\ p\ (\mathit{seq}\ q\ r) && \text{(composition)}\\
\mathit{seq}\ (\mathit{succeeds}\ f)\ (\mathit{succeeds}\ a) &= \mathit{succeeds}\ (f\ g) && \text{(homomorphism)}\\
\mathit{seq}\ p\ (\mathit{succeeds}\ a) &= \mathit{seq}\ (\mathit{succeeds}\ (\lambda f \to f\ a))\ p && \text{(interchange)}\\
\\
\mathit{seq}\ p\ (\mathit{alt}\ q\ r) &= \mathit{alt}\ (\mathit{seq}\ p\ q)\ (\mathit{seq}\ p\ r) && \text{(left-distributivity)}\\
\mathit{seq}\ (\mathit{alt}\ p\ q)\ r &= \mathit{alt}\ (\mathit{seq}\ p\ r)\ (\mathit{seq}\ q\ r) && \text{(right-distributivity)}
\end{aligned}
$$

Figure 5.2: Combinator laws for *alt* and *seq*.

Since f is a placeholder for any type, it can be a function type. If we ignore the semantic value f, then the definition corresponds exactly to the definition of *succeeds* in §3.1.1. The same holds for *seq*, and in §3.1.1 we showed that *seq* is associative with identity *succeeds*. Here *seq* is left-associative — as function application is left-associative — and indeed we can prove that $\mathit{seq}\ (\mathit{succeeds}\ \mathit{id})\ q = q$. In §5.2.2 we present additional laws satisfied by the combinators of this section.

The *alt* and *fails* combinators are defined as follows:

```
fails :: Parser t a
fails = Parser pf
  where pf inp k = []
alt :: Parser t a → Parser t a → Parser t a
alt (Parser p) (Parser q) = Parser pf
  where pf inp k = p inp k ++ q inp k
```

Two parse functions are only alternatives of each other if they produce the same type of semantic values (besides matching the same type of terminals).

5.2.2 Combinator Laws

Figure 5.2 shows a collection of laws that can be proven for *alt*. Compared to the laws of Figure 3.4 in §3.1.3, idempotency and commutativity no longer hold. This follows from

$$
\begin{aligned}
& \mathit{seq}\ (\mathit{succeeds}\ \mathit{id})\ q\ \mathit{inp}\ l \\
= {} & \mathit{concat}\ (\mathit{map}\ (\lambda(k,f) \to \mathit{map}\ (\lambda(r,a) \to (r, f\ a))\ (q\ \mathit{inp}\ k))\ (\mathit{succeeds}\ \mathit{id}\ \mathit{inp}\ l)) \\
= {} & \mathit{concat}\ (\mathit{map}\ (\lambda(k,f) \to \mathit{map}\ (\lambda(r,a) \to (r, f\ a))\ (q\ \mathit{inp}\ k))\ [(l, \mathit{id})]) \\
= {} & \mathit{concat}\ [(\lambda(k,f) \to \mathit{map}\ (\lambda(r,a) \to (r, f\ a))\ (q\ \mathit{inp}\ k))\ (l, \mathit{id})] \\
= {} & (\lambda(k,f) \to \mathit{map}\ (\lambda(r,a) \to (r, f\ a))\ (q\ \mathit{inp}\ k))\ (l, \mathit{id}) \\
= {} & \mathit{map}\ (\lambda(r,a) \to (r, \mathit{id}\ a))\ (q\ \mathit{inp}\ l) \\
= {} & \mathit{map}\ (\lambda(r,a) \to (r, a))\ (q\ \mathit{inp}\ l) \\
= {} & \mathit{map}\ \mathit{id}\ (q\ \mathit{inp}\ l) \\
= {} & q\ \mathit{inp}\ l
\end{aligned}
$$

Figure 5.3: Proof that *succeeds id* is a let identity of *seq* (see Figure 5.2.

$$
\begin{aligned}
& \mathit{seq}\ (\mathit{succeeds}\ f)\ (\mathit{succeeds}\ a)\ \mathit{inp}\ l \\
= {} & \mathit{concat}\ (\mathit{map}\ (\lambda(k,f) \to \mathit{map}\ (\lambda(r,a) \to (r, f\ a))\ [(k,a)])\ [(l,f)]) \\
= {} & \mathit{concat}\ (\mathit{map}\ (\lambda(k,f) \to [(\lambda(r,a) \to (r, f\ a))\ (k,a)])\ [(l,f)]) \\
= {} & \mathit{concat}\ [(\lambda(k,f) \to [(\lambda(r,a) \to (r, f\ a))\ (k,a)])\ (l,f)] \\
= {} & (\lambda(k,f) \to [(\lambda(r,a) \to (r, f\ a))\ (k,a)])\ (l,f) \\
= {} & [(\lambda(r,a) \to (r, f\ a))\ (l,a)] \\
= {} & [(l, f\ a)] \\
= {} & \mathit{succeeds}\ (f\ a)\ \mathit{inp}\ l
\end{aligned}
$$

Figure 5.4: Proof of the homomorphism property of Figure 5.2.

the usage of lists to collect multiple results, as lists are ordered (breaks commutativity) and may contain duplicates (breaks idempotency). The laws are trivial to prove, using the associativity of $(+\!\!+)$ to prove the associativity of *alt*.

Figure 5.2 also shows a collection of laws that can be proven for *seq*. Compared to the laws of Figure 3.4 in §3.1.3, right identity and associativity are lost. This follows from the left associativity of function application. The laws for absorption are easy to prove after replacing the use of list-comprehension in the definition of *seq* with the equivalent expression in terms of *map* and *concatMap*:

$$\mathit{seq}\ p\ q\ \mathit{inp}\ l = \mathit{concat}\ (\mathit{map}\ (\lambda(k,f) \to \mathit{map}\ (\lambda(r,a) \to (r, f\ a))\ (q\ \mathit{inp}\ k))\ (p\ \mathit{inp}\ l))$$

If in the definition above, either p or q yields the empty list, then the resulting list is empty. As examples, we prove that *succeeds id* is a left identity for *seq* in Figure 5.3 and we prove the homomorphism property in Figure 5.4. The composition and interchange properties can be proven in similar fashion.

```
instance Applicative (Parser t) where
  pure      = succeeds
  p ⟨*⟩ q   = seq p q
instance Alternative (Parser t) where
  empty     = fails
  p ⟨|⟩ q   = alt p q
instance Functor (Parser t) where
  fmap f p = pure f ⟨*⟩ p
```

Figure 5.5: *Applicative* and *Alternative* instances for *Parser t*.

5.2.3 An Applicative Interface

Haskell parser combinators are often defined as instances of the *Applicative* and *Alternative* type-classes, showing that parse functions are a member of a particular class of computations [Mcbride and Paterson, 2008]. This involves showing how the computations performed by parse functions can be done one after the other so that their results are somehow combined (by implementing ⟨∗⟩), how a value forms (the result of) a computation (by implementing *pure*), how a computation can be formed out of two alternatives (by implementing ⟨|⟩), what it means to have no alternatives (by implementing *empty*), and that certain laws hold on these functions and operators.

Based on the laws of Figure 5.2 we conclude that we can make *Parser t* (for all *t*) an instance of the *Applicative* type-class [Swierstra and Duponcheel, 1996] as well as the *Alternative* type-class. Figure 5.5 gives these instances. (Note that ⟨∗⟩ is a left-associative infix operator with higher priority than the associative infix operator ⟨|⟩.) The applicative type-class provides a number of useful methods[5], defined in terms of *pure* and ⟨∗⟩ (in other words, derived combinators). For example, in parser combinator terminology, operator ⟨$⟩ applies a semantic function to a parse function directly.

```
( ⟨$⟩ ) :: (a → b) → Parser t a → Parser t b
f ⟨$⟩ p = pure f ⟨*⟩ p    -- infix version of fmap
```

The combinators ⟨∗ and ∗⟩ are variants of ⟨∗⟩ used to ignore the semantic value of the right and left operand respectively.

[5]The definitions of these methods, as presented in this section, are available by default.

```
( ⟨∗ ) :: Parser t a → Parser t b → Parser t a
p ⟨∗ q = const ⟨$⟩ p ⟨∗⟩ q
( ∗⟩ ) :: Parser t a → Parser t b → Parser t b
p ∗⟩ q = flip const ⟨$⟩ p ⟨∗⟩ q
```

The following derived combinators apply a given parse functions zero or more and one or more times, respectively, combining the possibly many results in a list:

```
many, some :: Parser t a → Parser t [a]
some p = (:[]) ⟨$⟩ p ⟨|⟩ p ⟨∗⟩ some p   -- one repetition of p or more
many p = pure []    ⟨|⟩ some p          -- zero or more repetitions of p
```

With these combinators, it is easy to develop a parser for the grammar of Figure 4.1:

```
pE0    = (+) ⟨$⟩ pE1       ⟨∗ term '+' ⟨∗⟩ pE0      ⟨|⟩ pE1
pE1    = (∗) ⟨$⟩ pE2       ⟨∗ term '*' ⟨∗⟩ pE1      ⟨|⟩ pE2
pE2    =         term '(' ∗⟩ pE0       ⟨∗ term ')' ⟨|⟩ digits
digits = read ⟨$⟩ some digit   -- read interprets a string of digits as a number
digit  = foldr ( ⟨|⟩ ) empty (map term "01234567890")
```

The next section implements the explicit BNF combinators, and compares their functionality and usability with the combinators implemented here.

5.3 Explicit BNF Combinators

In this section we implement a BNF combinator library. The implementation is given in §5.3.3 and is based on the *internal* combinator libraries developed in §5.3.1 and §5.3.2. These combinator libraries are internal in the sense they are not exposed to the user and that their expressions are generated from, and with the same structure as, BNF combinator expressions. An internal expression of the first kind evaluates to a grammar, which is given to *fungll* to produce a parser. The parser is applied to a input sentence to yield a set of EPNs. An internal expression of the second kind builds a function that applies semantic functions based on derivation information extracted from the set of EPNs (Ridge's semantic phase).

The goal of this section is to define the core external BNF combinators *term*, ⟨::=⟩, *seqStart*, ⟨∗∗⟩, *altStart*, and ⟨|||⟩ in Section 5.3.3. The combinators *term* and ⟨::=⟩ construct symbol expressions (based on a terminal and alternates respectively), *seqStart* and ⟨∗∗⟩ construct sequence expressions, and *altStart* and ⟨|||⟩ construct choice expressions. The definition of combinator ⟨∗∗⟩ overrides the definition from Haskell's standard library.

```
gram_term    :: t → GramSymb l t
gram_nterm   :: (Ord t, Ord l) ⇒ l → GramCh l t → GramSymb l t
gram_altStart :: GramCh l t
gram_alt     :: GramCh l t → GramSeq l t → GramCh l t
gram_seqStart :: GramSeq l t
gram_seq     :: GramSeq l t → GramSymb l t → GramSeq l t
```

Figure 5.6: Signatures of the grammar combinators.

5.3.1 Grammar Combinators

In this subsection we implement a library of internal grammar combinators. The type signatures of the grammar combinators are given in Figure 5.6. The rich structure of the grammar combinator expressions makes it possible to generate grammars without binarisation.

Grammar generation In Ridge's combinator library P3 [Ridge, 2014], a unique nonterminal is generated for each combinator expression by combining the nonterminals generated for its subexpressions. The grammars generated this way are binarised in the sense that each nonterminal has one alternate with at most two symbols or two alternates with at most one symbol. When grammar descriptions are recursive, nonterminals need to be inserted to side-step nonterminal generation, thus avoiding nontermination[6]. We also rely on the insertion of nonterminals, via ⟨::=⟩ and *gram_nterm* internally, to avoid nontermination.

The types *GramSymb*, *GramSeq*, and *GramCh* are as follows:

```
type GramSymb l t = (Symbol l t, GrammarGen l t)
type GramSeq l t  = (Rhs l t, GrammarGen l t)
type GramCh l t   = ([Rhs l t], GrammarGen l t)
```

The first component of each combinator expression is the grammar fragment represented by it, e.g. the first component of a choice expression is the sequence of alternates it represents. The second component of each combinator expression is a 'grammar generator': a function that (possibly) extends a given grammar and a given set of nonterminals. The set of nonterminals is used to detect recursion and to avoid nontermination.

[6]This is a crude solution to the well-studied problem of making sharing observable in the implementation of embedded domain-specific languages [Claessen and Sands, 1999, Ljunglöf, 2002, Gill, 2009]. We consider it a pragmatic choice for our purposes.

```
gram_term t = (Term t, id)
gram_nterm n p = (Nt n, gen)
  where (alts, pgen) = p
        gen (nts, gram) | S.member n nts = (nts, gram)
                        | otherwise      = pgen (nts', gram')
          where nts' = S.insert n nts
                gram' = M.insertWith S.union n
                          (S.fromList alts) gram
gram_altStart = ([], id)
gram_alt (as, pgen) (α, qgen) = (as ++ [α], qgen ∘ pgen)
gram_seqStart = ([], id)
gram_seq (α, pgen) (s, qgen) = (α ++ [s], pgen ∘ qgen)
```

Figure 5.7: Definitions of the grammar combinators.

```
type GrammarGen n t =
       (S.Set n, Grammar n t) → (S.Set n, Grammar n t)
```

The definitions of *gram_term*, *gram_nterm*, *gram_altStart*, *gram_alt*, *gram_seqStart* and *gram_seq* are given in Figure 5.7 and correspond closely to the definitions in Section 4.2. Only *gram_nterm* extends the grammar with additional productions, one for each alternate represented by subexpression p. If the nonterminal n occurs in the set of nonterminals encountered so far (*nts*), no productions are added and the choice expression p is ignored. This is valid because the contribution of each nonterminal to the generated grammar is context-independent[7].

Grammars are obtained from symbol expressions by applying the function *grammarOf*. The generated grammars are augmented with a start symbol.

```
grammarOf :: n → GramSymb n t → Grammar n t
grammarOf start (n, pgen) = snd (pgen (S.empty, gram))
  where gram = M.singleton start (S.singleton [n])
```

Function *parserFor* applies *fungll* to the grammar generated for a symbol expression and produces a function from an input sentence to set of EPNs.

```
parserFor :: (Ord l, IsTerminal t) ⇒ l → GramSymb l t → Input t → EPNs l t
parserFor start p inp = snd (fungll (grammarOf start p) start inp)
```

[7]This assumes that nonterminals are inserted uniquely by the user.

```
type SemSymb n t a = (Symbol n t, OracleParser n t a)
type SemSeq n t a  = (Rhs n t, n → Rhs n t → OracleParser n t a)
type SemCh n t a   = (n → OracleParser n t a)
sem_term    :: t → SemSymb n t t
sem_nterm   :: (Ord n) ⇒ n → SemCh n t a → SemSymb n t a
sem_altStart :: SemCh n t a
sem_alt     :: SemCh n t a → SemSeq n t a → SemCh n t a
sem_seqStart :: a → SemSeq n t a
sem_seq     :: (Ord n, Ord t) ⇒ SemSeq n t (a → b) → SemSymb n t a → SemSeq n t b
```

Figure 5.8: The signatures of the semantic combinators.

5.3.2 Semantic Combinators

In this subsection we describe and implement semantic combinators, based on Ridge's semantic phase [Ridge, 2014], modifying it to our setting of non-binarised grammars (we omit memoisation). Ridge explains his combinators as parser combinators 'guided by an oracle', where the oracle — here a set of EPNs — provides a pre-computed set of pivots. The following type definition captures oracle-guided parsers:

```
type OracleParser n t a = Input t → EPNs n t → Int → Int → S.Set n → [a]
```

The signatures of the semantic combinators are given in Figure 5.8. Besides type variables n and t for nonterminals and terminals, the signatures also include the types of semantic values, similar to the parser combinators in Section 5.2. Besides a set of EPNs, an oracle-guided parser receives an input sentence (in order to yield elements of the input sentence as semantic values), a set of nonterminals, a left extent l and a right extent r. The set of nonterminals is used to detect recursion and avoid non-termination, as in Section 4.2. The left extent and right extent are used to select pivots from the EPN set. The pivots are obtained by applying the operation *pivots*:

```
pivots :: (Slot n t, Int, Int) → EPNs n t → [Int]
```

The semantic combinators compute grammar slots in order to apply *pivots*. The first component of a symbol expression is the symbol represented by the expression. Similarly, the first component of a sequence expression is the sequence of symbols represented by the expression. These are used to compute the required grammar slots. The argument of a

```
sem_term t = (Term t, gen)
  where gen inp ns l r nts | l + 1 ≡ r  = [inp ! l]
                           | otherwise = []
sem_nterm x p = (Nt x, gen)
  where gen inp ns l r nts | S.member x nts = []
                           | otherwise      = p x inp ns l r (S.insert x nts)
sem_alt p (α, q) = gen
  where gen x inp ns l r nts = p x inp ns l r nts ++ q x [] inp ns l r nts
sem_altStart = gen
  where gen x inp ns l r nts = []
sem_seqStart a = ([], gen)
  where gen x β inp ns l r nts | l ≡ r      = [a]
                               | otherwise = []
sem_seq (α, p) (s, q) = (α ++ [s], gen)
  where gen x β inp ns l r nts = [f a    | k ← pivots ((x, α ++ [s], β), l, r) ns
                                         , f ← p x (s : β) inp ns l k (leftLabels nts l k r)
                                         , a ← q inp ns k r (rightLabels nts l k r)]
        leftLabels nts l k r  | k < r      = S.empty
                              | otherwise = nts
        rightLabels nts l k r | k > l      = S.empty
                              | otherwise = nts
```

Figure 5.9: Definitions of the semantic combinators.

choice expression, and the arguments of the second component of a sequence expression, are used for the same purpose.

The semantics combinators are implemented in Figure 5.9. The semantic value of *sem_term* t is t', where t' is at position l in the input sentence (assuming that the EPN set determines that $matches\ t\ t' \equiv true$). The second component of a sequence expression receives nonterminal x and a list of symbols β as arguments. The arguments of *sem_seq* provide a list of symbols α and a symbol s (first component). This information is used to extract all the pivots $[k_1, \ldots, k_n]$ from the EPN set such that $((x, \alpha, \beta), l, k_i, r)$ is an EPN in the set, for all $1 \leqslant i \leqslant n$. For each k_i, p is applied with left extent l and right extent k_i to give semantic function $[f_1, \ldots, f_m]$ and q is applied with left extent k_i and right extent r to give semantic values $[a_1, \ldots, a_o]$. The semantic values of the sequence are the result of applying all f_i to all a_j, with $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant o$. Ambiguity reduction is required to keep the number of combinations under control.

The set *nts* is extended by x as part of the definition of *sem_nterm*, where x is a

nonterminal name, so that recursive calls can be detected. The set *nts* is emptied whenever a call is made to an oracle parser with a larger left extent, or a smaller right extent. Thus, if a recursive call is detected by inspecting *nts*, the call is guaranteed to be with the same left and right extent and it is implied that the nonterminal x can derive itself (according to the rules of the grammar described by the overarching combinator expression). If x derives a subsentence of the input, then it follows that there are infinitely many derivations of this subsentence, because the steps by which x derives itself can be repeated infinitely many times, each time giving a larger derivation (e.g. a BPT with more nodes). By terminating these recursive calls, we get only the interpretations of the smallest derivation, without the steps by which x derives itself. In Ridge's terminology, this means that 'bad' derivations are ignored [Ridge, 2014].

An evaluator is obtained by applying *evaluatorFor* to a symbol expression, an input sentence, initial left and right extents, and EPNs:

```
evaluatorFor :: SemSymb n t a → Input t → Int → Int → EPNs n t → [a]
evaluatorFor (_, p) inp l r ns = p inp ns l r S.empty
```

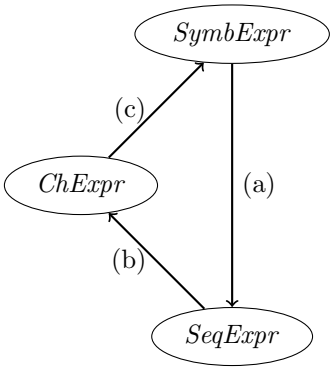
5.3.3 Flexible BNF Combinators

This subsection implements the BNF combinators by applying the internal combinators discussed in the previous subsections. A brief introduction to the BNF combinators was given at the start of this section, but the signatures of the combinators shown there were simplified. The actual signatures present a more general and flexible interface. The flexibility is achieved by automatic conversions between different types of combinator expressions. Using Haskell's type-classes, these conversions are realised as implicit coercions.

A BNF combinator expression is a pair of a grammar combinator expression (§5.3.1) and a semantic combinator expression (§5.3.2). The types *SymbExpr*, *ChExpr*, and *SeqExpr* are defined as follows:

```
newtype SymbExpr t a = Symb (GramSymb String t, SemSymb String t a)
newtype SeqExpr t a  = Seq (GramSeq String t, SemSeq String t a)
newtype ChExpr t a   = Ch (GramCh String t, SemCh String t a)
```

The type of nonterminals is no longer left abstract. Strings are chosen to support nonterminal generation.




```
class IsSeq seq where
  toSeq :: (Ord t, Show t) ⇒ seq t a → SeqExpr t a
instance IsSeq SeqExpr where
  toSeq = id
instance IsSeq SymbExpr where
  toSeq p = seqStart id ⟨**⟩ p    -- (a)
instance IsSeq ChExpr where
  toSeq = toSeq ∘ toSymb    -- (c) then (a)
class IsCh ch where
  toCh :: (Ord t, Show t) ⇒ ch t a → ChExpr t a
instance IsCh ChExpr where
  toCh = id
instance IsCh SeqExpr where
  toCh p = altStart ⟨||⟩ p          -- (b)
instance IsCh SymbExpr where
  toCh = toCh ∘ toSeq    -- (a) then (b)
class IsSymb symb where
  toSymb :: (Ord t, Show t) ⇒ symb t a → SymbExpr t a
instance IsSymb SymbExpr where
  toSymb = id
instance IsSymb ChExpr where
  toSymb p = genNt p ⟨::=⟩ p    -- (c)
instance IsSymb SeqExpr where
  toSymb = toSymb ∘ toCh    -- (b) then (c)
```

Figure 5.10: Conversions between types of expressions.

Consider the diagram in Figure 5.10. The edge labelled (a) represents a function *SymbExpr t a → SeqExpr t a*, converting symbol expressions into sequence expressions. Similarly, edge (b) represents a function *SeqExpr t a → ChExpr t a* and (c) represents a function *ChExpr t a → SymbExpr t a*. The core idea of this section is to implement these conversions as methods in a type-class and apply them in combinator definitions. For example, if p and q are symbol expressions, we can then write $p \langle||\rangle q$, which is automatically converted to *altStart* $\langle||\rangle$ p $\langle||\rangle$ q.

The type-classes and instances of Figure 5.10 implement the conversions (a), (b), and (c) of the diagram in Figure 5.10, as well as their compositions. To implement conversion (c), from a choice expression to a symbol expression, we use Ridge's technique, generating a string based on the structure of the choice expression. As shown by the following definitions, symbols are combined with `"*"` and alternates with `"|"`:

```
term :: (Show t) ⇒ t → SymbExpr t t
term t = Symb (gram_term t, sem_term t)
infixl 2 ⟨::=⟩
( ⟨::=⟩ ) :: (IsCh ch, Ord t, Show t) ⇒ String → ch t a → SymbExpr t a
l ⟨::=⟩ p = Symb (gram_nterm l pgram, sem_nterm l psem)
  where Ch (pgram, psem) = toCh p
infixl 4 ⟨**⟩
( ⟨**⟩ ) :: (IsSeq seq, IsSymb symb, Ord t, Show t) ⇒
             seq t (a → b) → symb t a → SeqExpr t b
p ⟨**⟩ q = Seq (gram_seq pgram qgram, sem_seq psem qsem)
  where Seq    (pgram, psem) = toSeq p
        Symb (qgram, qsem) = toSymb q
seqStart :: a → SeqExpr t a
seqStart a = Seq (gram_seqStart, sem_seqStart a)
infixl 3 ⟨|||⟩
( ⟨|||⟩ ) ::  (IsCh ch, IsSeq seq, Ord t, Show t) ⇒
               ch t a → seq t a → ChExpr t a
p ⟨|||⟩ q = Ch (gram_alt pgram qgram, sem_alt psem qsem)
  where Ch (pgram, psem) = toCh p
        Seq (qgram, qsem) = toSeq q
altStart :: ChExpr t a
altStart = Ch (gram_altStart, sem_altStart)
```

Figure 5.11: Flexible BNF combinators with coercions.

```
showRhs :: (Show n, Show t) ⇒ Rhs n t → String
showRhs []       = "__()"
showRhs (x : xs) = "__(" ++ show x ++ foldr comb "" xs ++ ")"
  where comb s acc = "*" ++ show s ++ acc
showRhss :: (Show n, Show t) ⇒ [Rhs n t] → String
showRhss []       = "__()"
showRhss (x : xs) = "__(" ++ showRhs x ++ foldr comb "" xs ++ ")"
  where comb s acc = "|" ++ showRhs x ++ acc
genNt :: (Show t) ⇒ ChExpr t a → String
genNt (Ch ((alts, _), _)) = showRhss alts
```

The BNF combinators are defined in Figure 5.11. Each combinator is defined as a straightforward application of the corresponding grammar combinator and semantic combinator to the subexpressions obtained by converting the combinator's arguments. The interface provided by the BNF combinators is flexible in the sense that the combinators ⟨::=⟩, ⟨|||⟩, and ⟨**⟩ can be applied to arbitrary BNF combinator expressions, as conversions between all combinator expressions are available. Perhaps the combinators are too flexible,

```
( ⟨::=⟩BIN ) :: (Ord t, Show t) ⇒ String → SymbExpr t a → SymbExpr t a
( ⟨::=⟩BIN ) = ( ⟨::=⟩ )
( ⟨||⟩BIN ) :: (Ord t, Show t) ⇒ SymbExpr t a → SymbExpr t a → SymbExpr t a
p ⟨||⟩BIN q = toSymb (p ⟨||⟩ q)
( ⟨**⟩BIN ) :: (Ord t, Show t) ⇒ SymbExpr t (a → b) → SymbExpr t a → SymbExpr t b
p ⟨**⟩BIN q = toSymb (p ⟨**⟩ q)
succeedsBIN :: (Ord t, Show t) ⇒ a → SymbExpr t a
succeedsBIN = toSymb ∘ seqStart
```

Figure 5.12: Binarising BNF combinators.

as recursive combinator expressions can be written without the use of ⟨::=⟩, thus causing nontermination. As a decision in the design of the library, we could have ignored conversion (c) — by omitting type-class *IsSymb* and the instances that involve *toSymb*, thereby forcing the user to insert nonterminal names manually with ⟨::=⟩. Instead, we decided to offer the most flexible interface. When aware of this risk, a user can avoid writing expressions involving conversion (c).

Function *execute* is given a start nonterminal, a symbol expression, and a sentence, and applies functions *parserFor* and *evaluatorFor* to yield all the interpretations of the sentence:

```
execute :: (IsTerminal t) ⇒ String → SymbExpr t a → [t] → [a]
execute start (Symb (pgram, psem)) str =
  evaluatorFor psem inp 0 (ub + 1) (parserFor start pgram inp)
  where (_, ub) = bounds inp
        inp     = listArray (0, length str − 1) str
```

5.3.4 Binarising BNF Combinators

The types of the flexible BNF combinators of the previous section are rather complex, in part due to their dependence on type-classes. Figure 5.12 defines variations on the core BNF combinators with much simpler types. As with conventional parser combinators, users encounter only one type of expressions — symbol expressions, in this case — when working with these combinators. The combinators are defined in terms of the flexible BNF combinators by applying *toSymb* to convert choice expressions and sequence expressions to symbol expressions where necessary. The grammars generated by the underlying grammar combinator expressions are binarised in the same sense as P3's internal grammars, because

of the explicit conversions with *toSymb*, and because of the implicit coercions on the symbol expression arguments, in the definitions of $\langle||\rangle_{\text{BIN}}$ and $\langle{*}{*}\rangle_{\text{BIN}}$. Taking the (explicit and implicit) conversions into account, the definitions of $\langle::=\rangle_{\text{BIN}}$, $\langle||\rangle_{\text{BIN}}$, and $\langle{*}{*}\rangle_{\text{BIN}}$ correspond directly to the definitions of *inject*, *alt*, and *seq* in Figure 4.3 of §4.2.1 respectively.

In Section 13.1 we use binarising BNF combinators to demonstrate the negative effects of grammar binarisation on the runtime efficiency of *fungll*.

Part II

Interpretation

Chapter 6

Transition System Semantics

This chapter introduces the topic of operational semantics for programming languages in the style of Plotkin (1970), commonly known as Structural Operational Semantics (SOS). Semantic specifications are written as a collection of inference rules that define transition relations over configurations, where a configuration provides an abstract model for the state of a machine executing a program. Transition relations formalise how a machine changes over time, as influenced by the program in its configuration. Other entities in the configuration influence the behaviour of the machine as well. For example the contents of the machine's store (also called heap) or any available input.

SOS specifications are not modular with respect to these auxiliary entities. If the structure of a configuration changes (for example to encode an additional entity), all inference rules given thus far need to be updated to reflect this change. This problem is addressed by Modular SOS (MSOS) introduced by Mosses (2004). MSOS is a generalisation of SOS in which each auxiliary entity is modelled by a category. The morphisms of the category specify how the entity can change as the machine transitions. In MSOS rules, an entity only needs to be mentioned if its transition deviates from the default specified by the composition operator of the category.

This chapter uses a simple imperative While language as a running example to explain the differences between SOS and MSOS specifications.

6.1 Transition Systems

This definition is due to [Plotkin, 2004b].

Definition 6.1.1. A Transition System (TS) is a structure $\langle \Gamma, \longrightarrow \rangle$ where Γ is a set of *configurations*, and $\longrightarrow \subseteq \Gamma \times \Gamma$ a *transition relation* ($\langle \gamma_1, \gamma_2 \rangle \in \longrightarrow$ is written $\gamma_1 \longrightarrow \gamma_2$). We call γ_1 the *source*, and γ_2 the *target*, of a transition $\gamma_1 \longrightarrow \gamma_2$. A *stuck configuration* is a configuration that does not appear as the source of any transition, i.e. $\gamma_1 \in \Gamma$ is stuck if for all $\gamma_2 \in \Gamma$ $\langle \gamma_1, \gamma_2 \rangle \notin \longrightarrow$.

An infinite computation from γ_1 in TS $\langle \Gamma, \longrightarrow \rangle$ is an infinite sequence of transitions $\gamma_1 \longrightarrow \gamma_1', \gamma_2 \longrightarrow \gamma_2', \ldots$ with $\gamma_i' = \gamma_{i+1}$ for all $i \geqslant 1$. A finite computation of length $n \geqslant 0$ from γ_1 to γ_{n+1} is a finite sequence of transitions $\gamma_1 \longrightarrow \gamma_2, \ldots, \gamma_n \longrightarrow \gamma_{n+1}$.

In the context of programming languages, it is useful to distinguish between successful and unsuccessful finite computations, under some notion of success. Successful termination of a computation is captured by terminal configurations, a nominated subset of stuck configurations that are considered the results of successful computations.

Definition 6.1.2. A Terminal Transition System (TTS) is a structure $\langle \Gamma, \longrightarrow, T \rangle$ where $\langle \Gamma, \longrightarrow \rangle$ is a TS, and $T \subseteq \Gamma$ a set of *terminal configurations* such that for all $\gamma_1 \in T$ and $\gamma_2 \in \Gamma$ it holds that $(\gamma_1, \gamma_2) \notin \longrightarrow$.

An infinite computation from γ_1 in TTS $\langle \Gamma, \longrightarrow, T \rangle$ is an infinite sequence of transitions $\gamma_1 \longrightarrow \gamma_1', \gamma_2 \longrightarrow \gamma_2', \ldots$ with $\gamma_i' = \gamma_{i+1}$ for all $i \geqslant 1$. A finite computation of length $n \geqslant 0$ from γ_1 to γ_{n+1} is a finite sequence of transitions $\gamma_1 \longrightarrow \gamma_2, \ldots, \gamma_n \longrightarrow \gamma_{n+1}$ with $\gamma_{n+1} \in T$.

Plotkin generalises terminal transition systems to allow for transitions with 'labels'.

Definition 6.1.3. A Labelled Terminal Transition System (LTTS) is a structure $\langle \Gamma, A, \rightarrow, T \rangle$ where Γ is a set of configurations, A is a set of *labels* (or actions), $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is a *labelled transition relation* ($\langle \gamma_1, \alpha, \gamma_2 \rangle \in \rightarrow$ is written $\gamma_1 \xrightarrow{\alpha} \gamma_2$), and T is a set of terminal configurations, such that for all $\gamma_1 \in T$ and $\gamma_2 \in \Gamma$ it holds that $(\gamma_1, \gamma_2) \notin \longrightarrow$.

A infinite computation from γ_1 in LTTS $\langle \Gamma, A, \rightarrow, T \rangle$ is an infinite sequence of transitions $\gamma_1 \xrightarrow{\alpha_1} \gamma_1', \gamma_2 \xrightarrow{\alpha_2} \gamma_2', \ldots$ with $\gamma_i' = \gamma_{i+1}$ for all $i \geqslant 1$. A finite computation of length $n \geqslant 0$ from γ_1 to γ_{n+1} is a finite sequence of transitions $\gamma_1 \xrightarrow{\alpha_1} \gamma_2, \ldots, \gamma_n \xrightarrow{\alpha_n} \gamma_{n+1}$ with $\gamma_{n+1} \in T$.

The *yield* of a sequence of labelled transitions $\gamma_1 \xrightarrow{\alpha_1} \gamma_2, \gamma_3 \xrightarrow{\alpha_2} \gamma_4, \ldots$ is the sequence of labels $\alpha_1, \alpha_2, \ldots$.

6.2 An Exemplary Language Definition

Plotkin demonstrates in his lecture notes how transition systems formalise different concepts in formal language theory such as finite automata and context-free grammars, as well as the semantics of programming language constructs [Plotkin, 2004b]. This section presents an operational semantics in the style of Plotkin for WHILE, a small imperative programming language with commands and expressions inspired by Astesiano's lecture notes [Astesiano, 1991]. The resulting specification does not have the modularity required for giving operational semantics to reusable components, a primary goal of this part of the thesis. In the next section we introduce MSOS and show that MSOS specifications do have the desired modularity.

6.2.1 Concrete and Abstract Syntax for While

The basis of our formal description of the syntax of WHILE is formed by the sets $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ of Booleans, $\mathbb{Z} = \{0, 1, 2, \ldots\} \cup \{-1, -2, \ldots\}$ of integers, and $Id = \{x, y, \ldots\}$ of identifiers. Following Plotkin's lecture notes [Plotkin, 2004b], we call these *basic syntactic sets*. The other syntactic objects of the language are elements of *derived syntactic sets*. The derived sets are defined inductively over basic and derived syntactic sets. To define derived syntactic sets, we use a particular form of grammar notation, inspired by the BNF-style of popular parser generator tools like Yacc. The grammar of WHILE is presented in Figure 6.1.

In this style, a grammar rule is the formal name of a set (e.g. E) followed by an informal name of the set (*expression*) and a number of alternatives with associated actions. Each grammar rule generates a nonterminal, and each alternative of the rule a production, of a context-free grammar. The context-free grammar describes the concrete syntax of the language formally. Moreover, we let each grammar rule define a derived syntactic set, by generating inference rules (abstract syntax rules) from the actions associated with each of the grammar rule's alternatives. The derived syntactic sets formally define the abstract

C	:	*command*	::=	C **;** C	$\{seq(C_1, C_2)\}$
			|	Id **:=** E	$\{assign(Id_1, E_1)\}$
			|	**print** E	$\{print(E_1)\}$
			|	**while** E **do** C **od**	$\{while(E_1, C_1)\}$
			|	D	$\{D_1\}$
D	:	*done*	::=	**done**	$\{done\}$
E	:	*expression*	::=	$E + E$	$\{plus(E_1, E_2)\}$
			|	$E \leqslant E$	$\{leq(E_1, E_2)\}$
			|	L	$\{L_1\}$
			|	Id	$\{Id_1\}$
L	:	*literal*	::=	$\mathbb{B}$	$\{\mathbb{B}_1\}$
			|	$\mathbb{Z}$	$\{\mathbb{Z}_1\}$

Figure 6.1: Grammar description for the WHILE language.

syntax of the language (at the end of this section we briefly discuss the distinction between concrete and abstract syntax).

For example, alternative $E + E$ defines a production $E ::= E + E$ in which E is a nonterminal (because it is the name of a derived syntactic set) and $+$ is a terminal symbol (because it is not the name of a derived set). The set of all symbols is thus inferable from the grammar description. The set of terminals is the union of all basic syntactic sets and (a representation of) all keywords (e.g. **print**, **od**, and $+$). The set of all nonterminals has a distinct element for each derived syntactic set.

The alternative $E + E$ has the action $\{plus(E_1, E_2)\}$ associated with it, generating the following abstract syntax rule:

$$\frac{Y_1 \in E \qquad Y_2 \in E}{plus(Y_1, Y_2) \in E}$$

A variable X_i in an action refers to the ith occurrence of the set X in the right-hand side of the alternative. The subscripts are thus used to form the bridge between concrete and abstract syntax (in a translation we do not formalise). The conditions of abstract syntax rules depend on sets mentioned in the action, e.g. the sets Id and E in the action $\{assign(Id_1, E_1)\}$ associated with the second alternative of *command*. In this example, the conclusion of the abstract syntax rule is $assign(Y_1, Y_2) \in C$. Here, C is the derived syntactic

set defined by the alternative. Therefore, this action generates the rule:

$$\frac{Y_1 \in Id \qquad Y_2 \in E}{assign(Y_1, Y_2) \in C}$$

As another example, the action associated with the first alternative of the grammar rule for L generates:

$$\frac{Y_1 \in \mathbb{B}}{Y_1 \in L}$$

Axioms are generated when no variables appear in the action. For example, the action $\{done\}$ generates the rule (axiom):

$$\frac{}{done \in D}$$

Together, the abstract syntax rules generated for each alternative of a grammar rule with formal name S define the derived syntactic set S.

Remarks on abstract syntax An element of abstract syntax is either an element of a basic syntactic set or the application of a *constructor* to zero or more syntactic elements. Constructor application is written in traditional functional style (e.g. $assign(S_1, S_2)$). Parentheses are omitted when a constructor is not applied to any elements (e.g. *done*). Constructors do not always have a fixed arity, i.e. some constructors are variadic.

An element of abstract syntax forms a tree called an *abstract syntax tree* (AST). An abstract syntax tree is an ordered tree with internal nodes labelled by constructor names, and leaf nodes labelled by a constructor name or an element from a basic syntactic set. Constructor application $F(S_1, \ldots, S_n)$ forms a leaf node labelled by F if $n = 0$, or an internal node labelled by F with the n children formed by $S_1, \ldots, S_n$. An element B of a basic syntactic set forms the leaf node labelled by B.

When discussing the semantics of programming languages, we take the abstract syntax of the language as a starting point. The informal inductive description of AST construction given above can be implemented by adding 'semantic actions' to parsers (as discussed in the context of parser combinators in Chapter 5). We can thus rely on the implementation of a

parser to provide an AST[1].

There is no clean-cut distinction between concrete syntax and abstract syntax. In fact, when describing a translation from programs of language S into language T it is often useful to gradually abstract over aspects of S, via several abstract syntaxes, and to subsequently introduce the concrete details of T. In this thesis we say that the (one and only) concrete syntax of a language is the set of productions from a context-free grammar for that language. The (one and only) abstract syntax of the language is a collection of basic syntactic sets and of *abstract syntax rules*: inference rules defining derived syntactic sets. Looking back at our example, one might say that the concrete syntax of WHILE is actually quite abstract, as it does not specify the precedence or associativity of operators.

6.2.2 Operational Semantics for While

The operational semantics of a WHILE program involves executing commands and evaluating expressions. Both types of operations change the state of the underlying machine by modifying its internal representation of the program. (This change is usually small.) The execution of a command may cause a change, a side-effect, to an other component of the machine. The evaluation of an expression is side-effect free, although it does depend on an other component of the machine (without changing it), namely to find the values assigned to identifiers.

An SOS specification models the state of a machine with a configuration: a tuple of elements containing the program, called the *program component* of the configuration. The other elements of a configuration are called *auxiliary entities*, modelling any additional components of the machine. The possible behaviours of a machine are modelled by the computations within a transition system. In the example of WHILE, we define two transition systems, for evaluating expressions and executing commands. A small-step style specification is given to introduce the small-step style with which funcons are defined in Chapter 8 and to enable a straightforward discussion on step size and computations towards the end of this section.

The behaviour of printing is to add a literal value (an element of the set L, obtained by evaluating an expression) to a sequence of values printed thus far (called the output of the

[1] In the case of a complete parser, this requires disambiguation.

$$\begin{array}{lcll} A & : & \texttt{out} & = & L^* \\ \sigma & : & \texttt{sto} & = & Id \to L \end{array}$$

Figure 6.2: Auxiliary entity sets for WHILE commands.

program). We follow the convention of modelling the storage of a machine as a function mapping identifiers to assigned values, called the store. The store and the program's output are auxiliary entities, formalised by the *auxiliary entity sets* of Figure 6.2. Here σ is the formal name of the set and `sto` is an informal name (later referred to as an entity identifier) associated with the set.

Small-step semantics for expressions We give a TTS for the evaluation of expressions by defining a transition relation via inference rules. The only auxiliary entity in a configuration is the store, for looking up assigned values. There is no output, and the store never changes; an expression has no side-effects. The transition relation $\longrightarrow_E \subseteq (E \times \sigma) \times (E \times \sigma)$ is defined in Figure 6.3. The meaning of $+$ and $\leqslant$ are assumed.

Note that we choose variable names merely to suggest that they are placeholders for elements of a certain set. Plotkin and Astesiano associate meaning with the chosen variable names so that conditions like $\mathbb{Z}_1 \in \mathbb{Z}$ can be left implicit. The conditions that do not involve a transition relation (also called side-conditions) are often written separately, 'besides the bar', from the conditions that do involve a transition relation (also called premises). Instead we mix both types of conditions and write them 'above the bar'.

The rules define a deterministic relation because each constructor in the abstract syntax has one or more mutually exclusive rules. For example, Rules (PLUS-1), (PLUS-2), and (PLUS) are mutually exclusive because an arbitrary expression E_1 cannot be literal (i.e. $E_1 \in L$) and be part of a transition simultaneously. These rules determine that the operands of *plus* (and similarly *leq*) are evaluated in a strictly left-to-right order, although small-step SOS is especially suitable to specify an interleaved evaluation of operands. An arbitrary evaluation order for the operands of *plus* can be specified by removing the condition $\mathbb{Z}_1 \in \mathbb{Z}$ from Rule (PLUS-2) (and for consistency also replacing $\mathbb{Z}_1$ with E_1). The resulting specification is no longer deterministic as Rules (PLUS-1) and (PLUS-2) are simultaneously applicable to all expressions $plus(E_1, E_2)$ for which E_1 and E_2 require (and permit) computation.

$$\frac{\langle E_1, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle}{\langle \mathit{plus}(E_1, E_2), \sigma_1\rangle \longrightarrow_E \langle \mathit{plus}(E_1', E_2), \sigma_1\rangle} \quad \text{(PLUS-1)}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \langle E_2, \sigma_1\rangle \longrightarrow_E \langle E_2', \sigma_1\rangle}{\langle \mathit{plus}(\mathbb{Z}_1, E_2), \sigma_1\rangle \longrightarrow_E \langle \mathit{plus}(\mathbb{Z}_1, E_2'), \sigma_1\rangle} \quad \text{(PLUS-2)}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \mathbb{Z}_2 \in \mathbb{Z} \qquad \mathbb{Z}_3 = \mathbb{Z}_1 + \mathbb{Z}_2}{\langle \mathit{plus}(\mathbb{Z}_1, \mathbb{Z}_2), \sigma_1\rangle \longrightarrow_E \langle \mathbb{Z}_3, \sigma_1\rangle} \quad \text{(PLUS)}$$

$$\frac{\langle E_1, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle}{\langle \mathit{leq}(E_1, E_2), \sigma_1\rangle \longrightarrow_E \langle \mathit{leq}(E_1', E_2), \sigma_1\rangle} \quad \text{(LEQ-1)}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \langle E_2, \sigma_1\rangle \longrightarrow_E \langle E_2', \sigma_1\rangle}{\langle \mathit{leq}(\mathbb{Z}_1, E_2), \sigma_1\rangle \longrightarrow_E \langle \mathit{leq}(\mathbb{Z}_1, E_2'), \sigma_1\rangle} \quad \text{(LEQ-2)}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \mathbb{Z}_2 \in \mathbb{Z} \qquad \mathbb{B}_1 = \mathbb{Z}_1 \leqslant \mathbb{Z}_2}{\langle \mathit{leq}(\mathbb{Z}_1, \mathbb{Z}_2), \sigma_1\rangle \longrightarrow_E \langle \mathbb{B}_1, \sigma_1\rangle} \quad \text{(LEQ)}$$

$$\frac{L_1 = \sigma_1(\mathit{Id}_1) \qquad \mathit{Id}_1 \in \mathit{Id}}{\langle \mathit{Id}_1, \sigma_1\rangle \longrightarrow_E \langle L_1, \sigma_1\rangle} \quad \text{(ASSIGNED)}$$

Figure 6.3: Small-step transition relation for WHILE expressions.

There are configurations for which no rule is applicable. For example, there is no transition from $\langle \mathit{plus}(\mathbb{B}_1, E_1), \sigma_1\rangle$, for any Boolean $\mathbb{B}_1$, expression E_1, and store σ_1. A less obvious example is $\langle \mathit{plus}(x, 3), \sigma_1\rangle$, for any identifier x and for any σ_1 that is not defined on x. Candidate (PLUS-1) is not applicable as there is no transition $\langle x, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle$ (for any E_1'), because the premise of rule (ASSIGNED) cannot be established. These configurations are not terminal configurations as they are not the result of successful computation. We say that terminal configurations must have a literal as a program component. The TTS for WHILE expressions is $\mathit{tts}_{\text{WHILE-EXPR}} = \langle E \times \sigma, \longrightarrow_E, L \times \sigma\rangle$.

Small-step semantics for commands A WHILE program is a single command, typically a binarised sequence of commands constructed by applications of *seq*. Whereas expressions are evaluated to obtain a value, commands are executed for their side-effects. To determine successful execution of a command, Plotkin uses a special kind of terminal configuration without a program component. Instead we follow Mosses, and introduce a distinctive com-

$$\frac{\langle C_1, \sigma_1, \alpha_1\rangle \longrightarrow_C \langle C_1', \sigma_2, \alpha_2\rangle}{\langle seq(C_1, C_2), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle seq(C_1', C_2), \sigma_2, \alpha_2\rangle} \quad \text{(SEQ)}$$

$$\langle seq(done, C_2), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle C_2, \sigma_1, \alpha_1\rangle \quad \text{(DONE)}$$

$$\frac{\langle E_1, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle}{\langle assign(Id_1, E_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle assign(Id_1, E_1'), \sigma_1, \alpha_1\rangle} \quad \text{(ASSIGN-2)}$$

$$\frac{L_1 \in L \qquad \sigma_2 = \sigma_1[Id_1 \mapsto L_1]}{\langle assign(Id_1, L_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle done, \sigma_2, \alpha_1\rangle} \quad \text{(ASSIGN)}$$

$$\frac{\langle E_1, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle}{\langle print(E_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle print(E_1'), \sigma_1, \alpha_1\rangle} \quad \text{(PRINT-1)}$$

$$\frac{L_1 \in L \qquad \alpha_2 = \alpha_1 +\!\!+ [L_1]}{\langle print(L_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle done, \sigma_1, \alpha_2\rangle} \quad \text{(PRINT)}$$

Figure 6.4: Small-step transition relation for WHILE commands (part 1).

mand, $done \in D$, to denote a fully executed command [Mosses, 2004]. Note that set D has a grammar rule in Figure 6.1, and *done* is therefore part of the concrete syntax of WHILE. If preferred, D can be defined in the abstract syntax only.

The configurations of the transition system for WHILE commands have stores and output as auxiliary entities. The small-step transition relation $\longrightarrow_C$ is defined for most commands in Figure 6.4.

Astesiano suggests several possible semantics for the *while* construct [Astesiano, 1991]. The difficulty of a small-step semantics is that, say $while(E_1, C_1)$, requires potentially many steps to evaluate the condition E_1. The command $seq(C_1, while(E_1, C_1))$ is to be executed when the condition evaluates to **true**, where E_1 is the condition before it was evaluated. Thus the original condition needs to be remembered. One of Astesiano's suggestion defers evaluation of the condition to an if-then-else command with $seq(C_1, while(E_1, C_1))$ as its then-branch, thus 'making a copy' of the condition. A second suggestion is to close $\longrightarrow_E$ under transitivity, effectively evaluating the condition in one step. A similar approach is discussed later. Rather than introducing if-then-else to the language, we allow the *while* constructor to take three arguments: two expressions and a command. The variant of *while*

$$\frac{Y_1 \in E \qquad Y_2 \in E \qquad Y_3 \in C}{\mathit{while}(Y_1, Y_2, Y_3) \in E} \qquad \text{(WHILE-AUX-SYNTAX)}$$

$$\frac{\langle \mathit{while}(E_1, E_1, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle C_2, \sigma_2, \alpha_2\rangle}{\langle \mathit{while}(E_1, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle C_2, \sigma_2, \alpha_2\rangle} \qquad \text{(WHILE-AUX)}$$

$$\frac{\langle E_1, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle}{\langle \mathit{while}(E_1, E_2, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle \mathit{while}(E_1', E_2, C_1), \sigma_1, \alpha_1\rangle} \qquad \text{(WHILE-1)}$$

$$\langle \mathit{while}(\mathbf{false}, E_2, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle \mathit{done}, \sigma_1, \alpha_1\rangle \qquad \text{(WHILE-FF)}$$

$$\langle \mathit{while}(\mathbf{true}, E_2, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle \mathit{seq}(C_1, \mathit{while}(E_2, E_2, C_1)), \sigma_1, \alpha_1\rangle \qquad \text{(WHILE-TT)}$$

Figure 6.5: Small-step transition relation for WHILE commands (part 2).

with three arguments is for 'internal' use only, storing a copy of the condition before it is evaluated. The first expression is the condition as it is being evaluated, the second expression is the copy of the condition prior to evaluation. The rules are given in Figure 6.5, including the abstract syntax extension (rule (WHILE-AUX-SYNTAX)). Rule (WHILE-AUX) defers the semantics of *while* with two arguments to the variant with three arguments by making a copy of the condition. Rule (WHILE-1) evaluates the first argument of *while*. Rule (WHILE-TT) shows that the original condition (E_2) is used to reinstate the *while* expression with the condition before it was evaluated.

Instead of writing rules (WHILE-AUX-SYNTAX) and (WHILE-AUX) we could use the following grammar rule alternative for *while* instead:

$$C \quad : \quad \mathit{command} \quad | \quad \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od} \quad \{\mathit{while}(E_1, E_1, C_1)\}$$

The TTS for executing commands is $tts_{\text{WHILE}} = \langle C \times \sigma \times A, \longrightarrow_C, D \times \sigma \times A\rangle$.

Remarks on step size The small-step transition relations $\longrightarrow_E$ and $\longrightarrow_C$ describe small steps indeed, and computations for non-trivial programs require many transitions. Consider the following computation:

$$\langle seq(assign(x, leq(plus(1, plus(2,3)), 7)), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C$$
$$\langle seq(assign(x, leq(plus(1,5), 7)), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C$$
$$\langle seq(assign(x, leq(6,7)), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C$$
$$\langle seq(assign(x, \mathbf{true}), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C$$
$$\langle seq(done, print(x)), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C$$
$$\langle print(x), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C$$
$$\langle print(\mathbf{true}), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C$$
$$\langle done, \sigma^0[x \mapsto \mathbf{true}], [\mathbf{true}]\rangle$$

The example is a computation in the TTS for commands and does not involve the transition relation $\longrightarrow_E$. The fact that the transition relation $\longrightarrow_C$ is defined in terms of $\longrightarrow_E$ is not visible in the computation. However, three steps are required to evaluate the subexpression $leq(plus(1, plus(2,3), 7))$. We may wish to 'hide' the steps that evaluate subexpressions from computations in the TTS for commands. One way to achieve this is closing $\longrightarrow_E$ under transitivity and reflexivity, adding the following rules:

$$\gamma_1 \longrightarrow_E \gamma_1 \qquad \frac{\gamma_1 \longrightarrow_E \gamma_2 \qquad \gamma_2 \longrightarrow_E \gamma_3}{\gamma_1 \longrightarrow_E \gamma_3}$$

Now $leq(plus(1, plus(2,3), 7)) \longrightarrow_E \mathbf{true}$ and the example computation shortens to:

$$\langle seq(assign(x, leq(plus(1, plus(2,3)), 7)), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C$$
$$\langle seq(assign(x, \mathbf{true}), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C$$
$$\langle seq(done, print(x)), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C$$
$$\langle print(x), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C$$
$$\langle print(\mathbf{true}), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C$$
$$\langle done, \sigma^0[x \mapsto \mathbf{true}], [\mathbf{true}]\rangle$$

The rule for reflexivity, however, violates the termination property of terminal transitions systems, i.e. there are transitions from terminal configurations. This is easily solved by adding a termination condition to the rule, restricting its applicability. The rule for transitivity makes a specification ambiguous, as potentially many proofs are possible of a single transition. This is because a sequence of transitions $\gamma_1 \longrightarrow_E \ldots \longrightarrow_E \gamma_n$ can be split in multiple ways to establish the premises of transitivity. Moreover, both rules make it difficult

to find proofs mechanically because they introduce nondeterminism to a specification. In particular, the rule for reflexivity is always applicable (without the termination condition) and the transitivity rule is applicable in many ways. For these reasons, we introduce a separate relation $\dashrightarrow_E$, capturing the finite computations in the TTS for WHILE expressions, defined as follows:

$$\frac{\gamma_1 \in L \times \sigma}{\gamma_1 \dashrightarrow_E \gamma_1} \tag{6.1}$$

$$\frac{\gamma_1 \longrightarrow_E \gamma_2 \qquad \gamma_2 \dashrightarrow_E \gamma_3}{\gamma_1 \dashrightarrow_E \gamma_3} \tag{6.2}$$

We refer to $\dashrightarrow_E$ as the iterative closure of $\longrightarrow_E$.

We replace rules (ASSIGN-2) and (ASSIGN) by:

$$\frac{\langle E_1, \sigma_1\rangle \dashrightarrow_E \langle L_1, \sigma_1\rangle \qquad \sigma_2 = \sigma_1[Id_1 \mapsto L_1]}{\langle assign(Id_1, E_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle done, \sigma_2, \alpha_1\rangle} \quad \text{(ASSIGN-EVAL)}$$

Since $leq(plus(1, plus(2, 3)), 7) \dashrightarrow_E \mathbf{true}$, the example computation shortens to:

$$\begin{array}{l}\langle seq(assign(x, leq(plus(1, plus(2, 3)), 7)), print(x)), \sigma^0, [\cdot]\rangle \longrightarrow_C \\ \langle seq(done, print(x)), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C \\ \langle print(x), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C \\ \langle print(\mathbf{true}), \sigma^0[x \mapsto \mathbf{true}], [\cdot]\rangle \longrightarrow_C \\ \langle done, \sigma^0[x \mapsto \mathbf{true}], [\mathbf{true}]\rangle\end{array}$$

The semantics of *while* loops is simplified by applying this method to evaluate loop conditions. Figure 6.6 shows the simplified semantics for *while* loops, removing the need for the variant of *while* with three arguments.

The style in which rules are written influences the structure of the computations, as well as that of the proofs showing these computations exist.

Remarks on notation The semantics of WHILE has been defined such that expressions do not modify the store. All transitions involving $\longrightarrow_E$, occurring in the rules defining $\longrightarrow_E$ and $\longrightarrow_C$, have the same store component in source and target. We can introduce syntactic

$$\frac{\langle E_1, \sigma_1\rangle \dashrightarrow_E \langle \mathbf{false}, \sigma_1\rangle}{\langle \mathit{while}(E_1, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle \mathit{done}, \sigma_1, \alpha_1\rangle} \quad \text{(WHILE-FF)}$$

$$\frac{\langle E_1, \sigma_1\rangle \dashrightarrow_E \langle \mathbf{true}, \sigma_1\rangle}{\langle \mathit{while}(E_1, C_1), \sigma_1, \alpha_1\rangle \longrightarrow_C \langle \mathit{seq}(C_1, \mathit{while}(E_1, C_1)), \sigma_1, \alpha_1\rangle} \quad \text{(WHILE-TT)}$$

Figure 6.6: Alternative semantics for while loops.

sugar to capture this common pattern and to improve the readability and maintainability of the rules. In the example of WHILE, we can let $\sigma_1 \vdash E_1 \longrightarrow_E E_1'$ denote $\langle E_1, \sigma_1\rangle \longrightarrow_E \langle E_1', \sigma_1\rangle$. A selection of simplified rules for WHILE expressions is given in Figure 6.7.

$$\frac{\sigma_1 \vdash E_1 \longrightarrow_E E_1'}{\sigma_1 \vdash \mathit{plus}(E_1, E_2) \longrightarrow_E \mathit{plus}(E_1', E_2)} \quad \text{(PLUS-1)}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \sigma_1 \vdash E_2 \longrightarrow_E E_2}{\sigma_1 \vdash \mathit{plus}(\mathbb{Z}_1, E_2) \longrightarrow_E \mathit{plus}(\mathbb{Z}_1, E_2')} \quad \text{(PLUS-2)}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \mathbb{Z}_2 \in \mathbb{Z} \qquad \mathbb{Z}_3 = \mathbb{Z}_1 + \mathbb{Z}_2}{\sigma_1 \vdash \mathit{plus}(\mathbb{Z}_1, \mathbb{Z}_2) \longrightarrow_E \mathbb{Z}_3} \quad \text{(PLUS)}$$

$$\frac{L_1 = \sigma_1(\mathit{Id}_1) \qquad \mathit{Id}_1 \in \mathit{Id}}{\sigma_1 \vdash \mathit{Id}_1 \longrightarrow_E L_1} \quad \text{(ASSIGNED)}$$

Figure 6.7: Variation of some rules in Figure 6.3 with syntactic sugar.

6.3 Generalised Transition Systems

The goal of the PLanCompS project has been to establish a formal and component-based approach to programming language development. The approach requires that the operational semantics of language constructs can be given in isolation whilst being composable. In an SOS specification, however, all inference rules that mention a particular transition relation have to be modified if we wish to change the structure of the configurations of the transition relation, for example to add or remove an auxiliary entity. A symptom of the

problem is observed in Figure 6.4. The rules for *assign* mention the unrelated output entity, and the rules for *print* mention the store.

To write modular rules we need the ability to omit references to auxiliary entities in rules for constructs independent of these auxiliary entities. When an auxiliary entity is added to a specification, the current set of rules should still be valid, although they do not mention the auxiliary entity. Simultaneously, the rules should specify how the new auxiliary entity should be *propagated* between the source and target of a transition and between transitions in a rule. In the next subsection we introduce Generalised Terminal Transition Systems (GTTSs), defined by MSOS rules that have these properties.

6.3.1 MSOS

Generalised transition systems have their basis in category theory. We follow [Mosses, 2004] and take a category to be: a set of objects O, a set of morphisms A, functions $source : A \to O$ and $target : A \to O$, a partial composition operator $\fatsemi : A \times A \to A$, and a function $id : O \to A$, giving an identity morphism to each object. The operator $\fatsemi$ and the identity morphisms are required to satisfy associativity and identity laws.

The following definition is by [Bach Poulsen, 2016] and is derived from [Mosses, 2004].

Definition 6.3.1. A Generalised Terminal Transition System (GTTS) is a structure $\langle \Gamma, \mathbb{C}, \longrightarrow, T \rangle$ where $\mathbb{C}$ is a category with morphisms A, such that $\langle \Gamma, A, \longrightarrow, T \rangle$ is an LTTS.

A computation in a GTTS is a computation in the underlying LTTS such that its yield is a path in the category $\mathbb{C}$: whenever a transition labelled a is followed immediately by a transition labelled a', the labels a and a' are required to be composable in $\mathbb{C}$.

[Mosses, 2004] introduces Modular SOS (MSOS) a modular variant of SOS in which MSOS rules define the transition relations of GTTSs. In MSOS, configurations are restricted to programs, and therefore only programs — not semantic entities — determine termination. Instead of in configurations, entities occur in the labels of transitions. For each entity a category is identified so that the morphisms of the category represent the possible changes to the entity. Identity morphisms represent 'no change', and the composition operator $\fatsemi$ restricts which changes can occur after each other. The categories of all entities are combined into a single product category. The product category forms a GTTS

together with a transition relation defined via MSOS rules. In MSOS rules, a variable can be a placeholder for a morphism in the product category, thus referring to entities without explicitly mentioning them. Conditions may include such *label variables* to restrict the ways in which they can be instantiated. For example, a condition $\mathit{unobs}(X)$ expresses that X must be a placeholder for a product of identity morphisms. A condition $X \fatsemi Y$ expresses that the morphisms in the products X and Y must compose. A transition in an MSOS rule is written as follows (where $n \geqslant 0$):

$$P_1 \xrightarrow{\{K_1=a_1,\ldots,K_n=a_n,X\}} P_2$$

In the *entity reference* $K_1 = a_1$, the left-hand side K_1 is the name of an entity (an index in an indexed product) and a_1 is a morphism. If the transition relation $\rightarrow$ has, as labels, morphisms from a J-indexed product category, then $\{K_1, \ldots, K_n\} \subseteq J$, and X is a label variable referring to a $(J \setminus \{K_1, \ldots, K_n\})$-indexed product of morphisms. For MSOS rules with constraints like $X \fatsemi Y$ to be well-defined, X and Y must be products with the same set of indices. Therefore, if entity K is mentioned explicitly in some transition of an MSOS rule, then all transitions within that rule must mention K explicitly. Moreover, all transition relations defined by an MSOS specification must involve the same entities (the same set of indices J). However, crucially, different MSOS rules may refer to different entities.

6.3.2 MSOS Specification of While

We revisit the running example and define GTTSs capturing the semantics of While expressions and commands and show that the GTTS for commands is modularly extensible.

Expressions The GTTSs share a $\{\mathtt{sto}, \mathtt{out}\}$-indexed product $\mathbb{C}$ of two preorder categories $\mathbb{C}_{\mathtt{sto}}$ and $\mathbb{C}_{\mathtt{out}}$ with object sets σ and A respectively. The GTTS for While expressions is $\langle E, \mathbb{C}, \rightarrow_E, L\rangle$ and the GTTS for commands is $\langle C, \mathbb{C}, \rightarrow_C, D\rangle$. The transition relation $\rightarrow_E$ is defined by the MSOS rules in Figure 6.8, together with the iterative closures of $\rightarrow_E$ and $\rightarrow_C$. When both arguments of $\mathit{plus}(\mathbb{Z}_1, \mathbb{Z}_2)$ are integers, then $\mathit{plus}(\mathbb{Z}_1, \mathbb{Z}_2)$ is replaced by the sum of the arguments, without any side-effects. The latter follows from the condition $\mathit{unobs}(X)$, specifying that X is an identity morphism. In the case of `sto` and `out`, this

$$\frac{\lambda_1 \in L \qquad unobs(X)}{\lambda_1 \overset{\{X\}}{\dashrightarrow}_E \lambda_1} \qquad \frac{E_1 \xrightarrow{\{X\}}_E E_1'}{plus(E_1, E_2) \xrightarrow{\{X\}}_E plus(E_1', E_2)}$$

$$\frac{\lambda_1 \xrightarrow{X}_E \lambda_2 \qquad \lambda_2 \overset{Y}{\dashrightarrow}_E \lambda_3 \qquad Z = X \fatsemi Y}{\lambda_1 \overset{Z}{\dashrightarrow}_E \lambda_3} \qquad \frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad E_2 \xrightarrow{\{X\}}_E E_2'}{plus(\mathbb{Z}_1, E_2) \xrightarrow{\{X\}}_E plus(\mathbb{Z}_1, E_2')}$$

$$\frac{unobs(X)}{done \overset{\{X\}}{\dashrightarrow}_C done} \qquad \frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \mathbb{Z}_2 \in \mathbb{Z} \qquad \mathbb{Z}_3 = \mathbb{Z}_1 + \mathbb{Z}_2 \qquad unobs(X)}{plus(\mathbb{Z}_1, \mathbb{Z}_2) \xrightarrow{\{X\}}_E \mathbb{Z}_3}$$

$$\frac{\gamma_1 \xrightarrow{\{X\}}_C \gamma_2 \qquad \gamma_2 \overset{\{Y\}}{\dashrightarrow}_C \gamma_3 \qquad Z = X \fatsemi Y}{\gamma_1 \overset{\{Z\}}{\dashrightarrow}_C \gamma_3} \qquad \frac{Id_1 \in Id \qquad L_1 = \sigma_1(Id_1) \qquad unobs(X)}{Id_1 \xrightarrow{\{\texttt{sto}=\langle\sigma_1,\sigma_1\rangle,X\}}_E L_1}$$

Figure 6.8: MSOS rules for $\dashrightarrow_E$, $\dashrightarrow_C$, and WHILE expressions (omitted *leq*).

means that the store is unchanged and that no values have been appended to the output. Importantly however, no assumptions are made about which entities are 'present'. The rules for evaluating the arguments of *plus* specify that if the evaluation of the argument has side-effects, then the evaluation of *plus* has the same side-effects (label variable X is used in both transitions). (Note that, unlike before, expressions may now have side-effects, if new forms of expressions are added to the language later.) The rules for *leq* are similar and have been omitted.

Since in preorder categories there is exactly one morphism a for every pair of objects $\langle o_1, o_2\rangle$, with $source(a) = o_1$ and $target(a) = o_2$, we can identify a by writing $\langle o_1, o_2\rangle$. An example is in the rule for evaluating identifiers, which involves inspecting the store to find values assigned to identifiers. An identifier evaluates without side-effects, following from the condition $unobs(X)$ and from the fact that $\langle\sigma_1, \sigma_1\rangle$ is an identity morphism.

Commands The MSOS rules that define $\rightarrow_C$ are given in Figure 6.9. To simplify computations in the GTTS for commands, we use single-step evaluation of subexpressions.

The rule for executing the first command in a sequence of commands specifies that the command may execute with side-effects and that these effects propagate as the effects of executing the sequence. A fully executed command is discarded from a sequence without

$$\frac{C_1 \xrightarrow{\{X\}}_C C_1'}{seq(C_1, C_2) \xrightarrow{\{X\}}_C seq(C_1', C_2)} \quad \text{(SEQ-CONG)}$$

$$\frac{unobs(X)}{seq(done, C_2) \xrightarrow{\{X\}}_C C_2} \quad \text{(SEQ-DONE)}$$

$$\frac{E_1 \overset{\{X\}}{\dashrightarrow}_E \mathbf{false}}{while(E_1, C_1) \xrightarrow{\{X\}}_C done} \quad \text{(WHILE-FALSE)}$$

$$\frac{E_1 \overset{\{X\}}{\dashrightarrow}_E \mathbf{true}}{while(E_1, C_1) \xrightarrow{\{X\}}_C seq(C_1, while(E_1, C_1))} \quad \text{(WHILE-TRUE-A)}$$

$$\frac{E_1 \overset{\{\texttt{sto}=\langle\sigma_1,\sigma_2\rangle,X\}}{\dashrightarrow}_E L_1 \qquad \sigma_3 = \sigma_2[Id_1 \mapsto L_1]}{assign(Id_1, E_1) \xrightarrow{\{\texttt{sto}=\langle\sigma_1,\sigma_3\rangle,X\}}_C done} \quad \text{(ASSIGN)}$$

$$\frac{E_1 \overset{\{\texttt{out}=\langle\alpha_1,\alpha_2\rangle,X\}}{\dashrightarrow}_E L_1 \qquad \alpha_3 = \alpha_2 +\!\!+ [L_1]}{print(E_1) \xrightarrow{\{\texttt{out}=\langle\alpha_1,\alpha_3\rangle,X\}}_C done} \quad \text{(PRINT)}$$

Figure 6.9: MSOS rules for WHILE commands.

side-effects. A step to evaluate the expression of an assignment may have side-effects which are propagated as the side-effects of the assignment. As an additional effect, the store is updated so that the first argument (an identifier) refers to the value of the second argument. The rule for *print* is similar to that of *assign* but acts on the output rather than the store. Note that the rules for *assign* do not mention `out` and the rules for *print* do not mention `sto`. The rules for the *while* construct follow the same pattern: any side-effects resulting from the evaluation of the condition of a loop propagate as the side-effects of the loop.

Extending While

We demonstrate the modularity of MSOS specifications by showing how WHILE is extended, without the need to revisit the rules written so far. As an example we add the *continue* command. Informally, the semantics of *continue* are to jump to the next iteration of the

(innermost) loop in which it occurs. The grammar fragment below provides a modular extension to the concrete and abstract syntax defined by the grammar in Figure 6.1.

$$C \quad : \quad \mathit{command} \quad ::= \quad \textbf{continue} \quad \{\mathit{continue}\}$$

To define the semantics of *continue* formally, we introduce an additional auxiliary entity by defining the following auxiliary entity set:

$$\mathit{Sig} \quad : \quad \texttt{sig} \quad = \quad \{\textbf{none}, \textbf{cnt}\}$$

Informally, a **cnt** signal is 'emitted' when a *continue* command is executed. (By default, transitions emit the signal **none**.) The body of a loop is wrapped inside a special constructor *listen_cnt* for responding to **cnt** signals. This is the only reason *listen_cnt* exists and it has no counterpart in the concrete syntax (*listen_cnt* is part of the *auxiliary abstract syntax*).

$$\frac{C_1 \in C}{\mathit{listen\_cnt}(C_1) \in C}$$

The auxiliary entity set *Sig* provides the objects of a new category $\mathbb{C}_{\texttt{sig}}$, an instance of Bach Poulsen's abrupt termination category [Bach Poulsen, 2016]. The product category $\mathbb{C}$ is extended with index `sig`. The morphisms of the category $\mathbb{C}_{\texttt{sig}}$ are $\langle \textbf{none}, \textbf{none} \rangle$, $\langle \textbf{none}, \textbf{cnt} \rangle$, and $\langle \textbf{cnt}, \textbf{cnt} \rangle$. The set of morphisms is closed under composition, and the identity morphisms are $\langle \textbf{none}, \textbf{none} \rangle$ and $\langle \textbf{cnt}, \textbf{cnt} \rangle$. Despite the change to $\mathbb{C}$, the rules given earlier in Figures 6.8 and 6.9 are still valid and meaningful.

The semantics of *continue* is given in Figure 6.10. Note that rule (WHILE-TRUE-B) is a replacement for (WHILE-TRUE-A). The rule shows that the body of a while loop is 'wrapped' within *listen_cnt* before it is executed. The continue command transition to *done*, accompanied by a **cnt** signal. Command *listen_cnt* is fully executed once its argument is fully executed or once its argument emits a **cnt** signal.

Only programs in which a *continue* command occurs outside a loop cause the morphisms $\langle \textbf{none}, \textbf{cnt} \rangle$ and $\langle \textbf{cnt}, \textbf{cnt} \rangle$ to appear in computations. A static semantics for WHILE might determine that such programs are not valid.

$$\frac{E_1 \xrightarrow{\{X\}}_E \mathbf{true}}{\mathit{while}(E_1, C_1) \xrightarrow{\{X\}}_C \mathit{seq}(\mathit{listen\_cnt}(C_1), \mathit{while}(E_1, C_1))} \quad \text{(WHILE-TRUE-B)}$$

$$\frac{\mathit{unobs}(X)}{\mathit{continue} \xrightarrow{\{\mathtt{sig}=\langle \mathbf{none}, \mathbf{cnt}\rangle, X\}}_C \mathit{done}} \quad \text{(CONTINUE)}$$

$$\frac{C_1 \xrightarrow{\{\mathtt{sig}=\langle \mathbf{none}, \mathbf{none}\rangle, X\}}_C C_1'}{\mathit{listen\_cnt}(C_1) \xrightarrow{\{\mathtt{sig}=\langle \mathbf{none}, \mathbf{none}\rangle, X\}}_C \mathit{listen\_cnt}(C_1')} \quad \text{(LISTEN-CNT-NONE)}$$

$$\frac{C_1 \xrightarrow{\{\mathtt{sig}=\langle \mathbf{none}, \mathbf{cnt}\rangle, X\}}_C C_1'}{\mathit{listen\_cnt}(C_1) \xrightarrow{\{\mathtt{sig}=\langle \mathbf{none}, \mathbf{none}\rangle, X\}}_C \mathit{done}} \quad \text{(LISTEN-CNT-CNT)}$$

$$\frac{\mathit{unobs}(X)}{\mathit{listen\_cnt}(\mathit{done}) \xrightarrow{\{X\}}_C \mathit{done}} \quad \text{(LISTEN-CNT-DONE)}$$

Figure 6.10: MSOS rules for WHILE extended with the *continue* command.

6.3.3 I-MSOS Rules

In the MSOS rules given for WHILE in the previous subsection, the usage of label variables and the conditions $\mathit{unobs}(X)$ and $X \wr Y$ follow a certain pattern. When a rule is written without a premise (e.g. rules (CONTINUE), and (SEQ-DONE)), the rule includes $\mathit{unobs}(X)$ as a side-condition, where X is the label variable of the conclusion. Thus, entities not mentioned in axioms do not change. When a rule is written with one premise (e.g. rules (SEQ-CONG) and (WHILE-TRUE-B)), then the same label variable is used in both premise and conclusion.

The described pattern is common for programming language definitions. This observation is at the heart of I-MSOS rules [Mosses and New, 2009], a variation on MSOS rules. In I-MSOS rules, there are no explicit label variables. Instead, label variables and conditions on label variables are left implicit (I-MSOS stands for Implicitly-Modular SOS). For example, rule (PRINT) can be replaced by the following I-MSOS rule:

$$\frac{E_1 \longrightarrow_E L_1 \qquad \alpha_2 = \alpha_1 +\!\!+ [L_1]}{\langle \mathit{print}(E_1), \mathtt{out} = \alpha_1 \rangle \longrightarrow_C \langle \mathit{done}, \mathtt{out} = \alpha_2 \rangle}$$

Entity references are no longer written on top of transition arrows. As a result, I-MSOS rules look similar to conventional SOS rules. Entity references are written differently depending on how the morphisms of the entity's category can be identified. We shall see more examples when we consider the I-MSOS rules of the CBS meta-language in Chapter 8.

I-MSOS rules define GTTSs indirectly, through underlying MSOS. Depending on the number of premises in an I-MSOS rule, different label variables and conditions are used in the underlying MSOS rule. To explain precisely what MSOS rules underlie I-MSOS rules, we let k be the number of premises in some I-MSOS rule. In the underlying MSOS rule, we take X_0 for the conclusion's label variable and X_i for the label variable of the ith premise. We explain, based on different values of k, which side conditions are active in the underlying MSOS rule. When $k = 0$, the underlying MSOS rule has the condition $unobs(X_0)$. When $k = 1$, we have $X_0 = X_1$ in the underlying MSOS rule. When $k > 1$, the underlying MSOS rule has the side condition $X_0 = X_1 \fatsemi \ldots \fatsemi X_k$. The latter guarantees that all changes in entity values represented by the morphisms in $X_1 \ldots X_k$ are composable and that their effects accumulate to form the effects of the conclusion. Since $\fatsemi$ is not commutative, this 'implicit propagation scheme' implies that I-MSOS rules are not truly declarative; the order in which premises are written matters. This topic is discussed in more detail in [Mosses and New, 2009], together with alternative implicit propagation schemes affecting the case $k > 1$. As an example of an I-MSOS rule with multiple premises, consider the following rule, (partially) defining the iterative closure of some $\longrightarrow$.

$$\frac{\lambda_1 \longrightarrow \lambda_2 \qquad \lambda_2 \dashrightarrow \lambda_3}{\lambda_1 \dashrightarrow \lambda_3}$$

I-MSOS rules may seem significantly less expressive than MSOS rules as they do not have explicit label variables. However, I-MSOS rules have proven to be widely applicable in the context of programming language semantics [Churchill et al., 2015, Sculthorpe et al., 2016], and are certainly sufficient for the purposes of this thesis.

Chapter 7

Modular Rule-Based Semantics

The previous chapter shows how MSOS specifications give semantics to programming language constructs with a high degree of modularity. This chapter introduces modular rule-based semantics (MRBS) inspired by MSOS. A detailed comparison between MRBS and MSOS is made at the end of §7.1.1.

The transition systems of SOS and MSOS are very abstract and they can be used to model more than programming language semantics. In particular, transitions systems do not mention auxiliary entities or program fragments and they can be defined via other means than inference rules. MRBS is designed specifically for the purposes of this thesis and formalises transition relations over configurations consisting of program fragments and auxiliary entities. Most importantly, MRBS has a notion of derivations based on production rules. Using the notion of derivations, we formalise interpreters as algorithms that find derivations, similar to the parsers of Chapter 2. Modularity is achieved by allowing derivations to refer to entities that do not occur in production rules when they satisfy certain propagation conditions.

This chapter also introduces a meta-language called IML for writing MRBS specifications, and shows how interpreters are generated from IML programs. IML is designed to enable mechanical treatment on the one hand. On the other hand, it is intended to be sufficiently general so that many operational aspects of deterministic programming languages can be specified and that different 'styles' of rules can be written.

7.1 Modular Rule-Based Specifications

A Modular Rule-Based Semantic Specification (MRBS) defines multiple transition relations such that each defines its own Terminal Transition System (TTS). The transitions are over configurations whose program components are taken from an abstract set of program fragments. All transition relations use the same auxiliary entities. Program fragments are considered terminating or not relative to a particular relation. That is, a program fragment may be considered the result of a computation according to one relation but not according to another. Entities determine termination as well as program fragments. A configuration is considered terminal with respect to a relation $\longrightarrow$ if its program component is terminating with respect to $\longrightarrow$, or if any of its auxiliary entities is considered terminating (the termination property of an entity is not specific to a particular relation). These aspects of MRBS specifications are captured by the following definitions.

Definition 7.1.1. A *signature* Σ consists of a set of program fragments Λ_Σ, a set of relation symbols R_Σ, a set of entity identifiers E_Σ, with R_Σ and E_Σ distinct, and a family of sets of program terminals $\{\Lambda^T_{\Sigma,k}\}_{k\in(R_\Sigma\cup E_\Sigma)}$ with $\Lambda^T_{\Sigma,k} \subseteq \Lambda_\Sigma$ for each $k \in R_\Sigma \cup E_\Sigma$.

The subscript Σ is omitted when it is clear from context to which signature is referred.

A configuration is formalised as a pair of a program component and an entity record, a partial function mapping entity identifiers to program fragments.

Definition 7.1.2. Given a signature Σ, a *configuration* γ is a pair $\langle\lambda, \delta\rangle$ with $prgm(\gamma) = \lambda \in \Lambda$ (the configuration's program component), $rec(\gamma) = \delta \in E \rightharpoonup \Lambda$ (the configuration's entity record), and $dom(\gamma) = dom(\delta) = \{j \in E \mid \delta(j)$ **defined**$\}$ (the domain of the entity record and the configuration). A configuration γ is *complete* if $dom(\gamma) = E$. A configuration γ is *terminal* in $r \in R$ if $prgm(\gamma) \in \Lambda^T_r$ or if there is an entity identifier $j \in dom(\gamma)$ with $rec(\gamma)(j) \in \Lambda^T_j$. Two configurations $\gamma = \langle\lambda, \delta\rangle$ and $\gamma' = \langle\lambda', \delta'\rangle$ are similar, written as $\gamma \simeq \gamma'$, if and only if for all $j \in (dom(\gamma) \cap dom(\gamma'))$ we have that $\delta(j) = \delta'(j)$ and if we have that $\lambda = \lambda'$.

A transition is a triple of two configurations γ_1, γ_2, and a relation symbol r, such that the configurations have the same domain and such that γ_1 is not terminal in r.

Definition 7.1.3. Given a signature Σ, a *transition* p_0 is a triple $\langle \gamma_1, r, \gamma_2 \rangle$, with γ_1 and γ_2 configurations such that $dom(\gamma_1) = dom(\gamma_2)$ and γ_1 not terminal in r, $src(p_0) = \gamma_1$ (the source of the transition), $tgt(p_0) = \gamma_2$ (the target of the transition), $rel(p_0) = r \in R$ (the relation symbol of the transition) and $dom(p_0) = dom(\gamma_1)$ (the domain of the transition). A transition p_0 is *complete* if $dom(p_0) = E$. Two transitions $p = \langle \gamma_1, r, \gamma_2 \rangle$ and $p' = \langle \gamma_1', r', \gamma_2' \rangle$ are similar, written as $p \simeq p'$, if and only if we have that $\gamma_1 \simeq \gamma_1'$, $\gamma_2 \simeq \gamma_2'$ and $r = r'$.

Inspired by context-free grammars, we define productions. A production has a conclusion and zero or more premises, each a transition with possibly different relation symbols.

Definition 7.1.4. Given a signature Σ, a *production* ψ is a pair $\langle p_0, \rho \rangle$, with p_0 a transition, $\rho = \{p_1, \ldots, p_k\}$ a (possibly empty) set of transitions such that $dom(p_0) = dom(p_i)$, for all $1 \leqslant i \leqslant k$, $conc(\psi) = p_0$ (the production's conclusion), $prem(\psi) = \rho$ (the production's premises), $rel(\psi) = rel(p_0)$ (the relation (partially) defined by ψ), and $dom(\psi) = dom(p_0)$ (the domain of the production). A production ψ is *complete* if $dom(\psi) = E$.

Definition 7.1.5. A *rule-based specification* Φ consists of a set of program fragments Λ_Φ, a set of relation symbols R_Φ, a set of entity identifiers E_Φ, a family of sets of program terminals $\{\Lambda^T_{\Phi,k}\}_{k \in (R_\Phi \cup E_\Phi)}$ (together forming a signature), and a set of productions Ψ_Φ.

The subscript Φ is omitted when it is clear from context to which specification is referred.

The semantics of a language is expressed by the possible transitions over configurations for which derivations can be formed by applying productions. We formalise the notion of a derivation tree, enabling us to discuss interpreters that find derivations automatically.

Definition 7.1.6. Given a specification Φ, a *derivation* t_0 is a pair $\langle s_0, t_1 \ldots t_k \rangle$, with $root(t_0) = s_0$ a transition (the root of the derivation), $chn(t_0) = t_1 \ldots t_k$ a sequence of derivations (the children of t_0), and $t_i = \langle s_i, c_i \rangle$ with $dom(s_0) = dom(s_i)$, for all $1 \leqslant i \leqslant k$, for which there is a production $\langle p_0, \{p_1, \ldots, p_k\} \rangle \in \Psi$ (the production producing the derivation) with $J = dom(s_0) \subseteq E$, $J^M = dom(p_0) \subseteq J$, $J^U = J \setminus J^M$ and $\forall_{0 \leqslant i \leqslant k}(p_i \simeq s_i)$ such that:

$$\forall_{j \in J^U} \quad k = 0 \implies rec(src(s_0))(j) = rec(tgt(s_0))(j) \qquad \text{(unobservability-prop)}$$

$$\forall_{j \in J^U} \quad k > 0 \implies rec(src(s_0))(j) = rec(src(s_1))(j) \qquad \text{(conclusion-prop(1))}$$

$$\forall_{j \in J^U} \quad k > 0 \implies rec(tgt(s_k))(j) = rec(tgt(s_0))(j) \qquad \text{(conclusion-prop(2))}$$

$$\forall_{j \in J^U, 1 \leqslant i < k} \quad k > 1 \implies rec(tgt(s_i))(j) = rec(src(s_{i+1}))(j) \qquad \text{(premises-prop)}$$

The domain of a derivation t_0 is the domain of its root, i.e. $dom(t_0) = dom(root(t_0))$. A derivation t_0 is *complete* if $dom(t_0) = E$. A configuration γ is stuck in $r \in R$ if there is no derivation t_0 with root $\langle \gamma_1, r, \gamma_2 \rangle$ and $dom(\gamma) \subseteq dom(t_0)$ such that $\gamma \simeq \gamma_1$. It follows from the definition of productions and transitions that all configurations terminal in $r \in R$ are stuck in r.

The root of a derivation often refers to entities that are not mentioned in the production producing that derivation. The definition of derivations relies on the similarity of transitions to relate the transitions in derivations to the transitions in productions. To restrict the unreferenced entities (J^U), the definition of derivations specifies how any unreferenced entities must be 'propagated'. The constraints placed on unmentioned entities are based on the implicit propagation scheme for entities in I-MSOS, discussed in §6.3.3.

As explained in §6.2.2, we use the iterative closure $\dashrightarrow$ of a transition relation $\longrightarrow$ to capture the finite computations involving $\longrightarrow$. The relation $\dashrightarrow$ can be defined within our framework. We can therefore prove the existence of finite computations using our notion of derivations. Proving the existence of infinite computations may be possible with co-inductive reasoning [Gordon, 1995], but this is not considered in this thesis.

We formalise interpreters as algorithms finding one or more derivations.

Specification 1. An algorithm is a *rule-based interpreter* if given a rule-based specification Φ, a configuration γ_1 and a relation symbol $r \in R$, it returns a derivation with a root of the form $\langle \gamma_1, r, \gamma_2 \rangle$ if there is any.

Specification 2. An algorithm is a *complete rule-based interpreter* if given a rule-based specification Φ, a configuration γ_1 and a relation symbol $r \in R$, it returns the set T of all derivations with a root of the form $\langle \gamma_1, r, \gamma_2 \rangle$.

The purpose of the next sections of this chapter is to explain the IML meta-language in which MRBS specifications are defined, considering again the WHILE example, and to formalise how rule-based interpreters are generated for, and invoked by, IML programs.

Remark on the connection with transition systems A rule-based specification Φ defines a TTS $\langle \Gamma_r, \longrightarrow_r, \Gamma_r^T \rangle$ for each relation symbol $r \in R$ with $\Gamma_r = \Lambda \times (E \to \Lambda)$, $\Gamma_r^T = \{\gamma \mid \gamma \in \Gamma_r, \gamma \textbf{ terminal in } r\}$, and $\gamma_1 \longrightarrow_r \gamma_2$ if and only if there is a complete derivation

$$\frac{\lambda_1 \in L}{\{\lambda_1\} \dashrightarrow_E \{\lambda_1\}}$$

$$\frac{\{\lambda_1\} \longrightarrow_E \{\lambda_2\} \qquad \{\lambda_2\} \dashrightarrow_E \{\lambda_3\}}{\{\lambda_1\} \dashrightarrow_E \{\lambda_3\}}$$

$$\{done\} \dashrightarrow_C \{done\}$$

$$\frac{\{\gamma_1\} \longrightarrow_C \{\gamma_2\} \qquad \{\gamma_2\} \dashrightarrow_C \{\gamma_3\}}{\{\gamma_1\} \dashrightarrow_C \{\gamma_3\}}$$

$$\frac{\{E_1\} \longrightarrow_E \{E_1'\}}{\{plus(E_1, E_2)\} \longrightarrow_E \{plus(E_1', E_2)\}}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \{E_2\} \longrightarrow_E \{E_2'\}}{\{plus(\mathbb{Z}_1, E_2)\} \longrightarrow_E \{plus(\mathbb{Z}_1, E_2'\}}$$

$$\frac{\mathbb{Z}_1 \in \mathbb{Z} \qquad \mathbb{Z}_2 \in \mathbb{Z} \qquad \mathbb{Z}_3 = \mathbb{Z}_1 + \mathbb{Z}_2}{\{plus(\mathbb{Z}_1, \mathbb{Z}_2)\} \longrightarrow_E \{\mathbb{Z}_3\}}$$

$$\frac{Id_1 \in Id \qquad L_1 = \sigma_1(Id_1)}{\{Id_1, \texttt{sto} = \sigma_1\} \longrightarrow_E \{L_1, \texttt{sto} = \sigma_1\}}$$

Figure 7.1: Rules for $\dashrightarrow_E$, $\dashrightarrow_C$ and for WHILE expressions (omitting *leq*).

with root $\langle \gamma_1, r, \gamma_2 \rangle$ in Φ, for all $\gamma_1, \gamma_2 \in \Gamma_r$. We have that $\gamma_1 \longrightarrow_r \gamma_2 \implies \gamma_1 \notin \Gamma_r^T$ as there is no derivation with root $\langle \gamma_1, r, \gamma_2 \rangle$ if γ_1 is terminal in r.

7.1.1 MRBS Specification for While

We revisit the running example and define an MRBS specification Φ for WHILE.

The productions of Φ are defined through inference rules in the following manner. In inference rules, a configuration is written as follows:

$$\{\lambda_0, j_1 = \lambda_1, \ldots, j_m = \lambda_m\}$$

This denotes the configuration $\langle \lambda_0, \delta \rangle$ with domain $\{j_1, \ldots, j_m\}$ and $\delta(j_i) = \lambda_i$ for all $1 \leqslant i \leqslant m$. A transition $\langle \gamma_1, \longrightarrow, \gamma_2 \rangle$ is written as:

$$\{\lambda_0^1, j_1 = \lambda_1^1, \ldots, j_m = \lambda_m^1\} \longrightarrow \{\lambda_0^2, j_1 = \lambda_1^2, \ldots, j_m = \lambda_m^2\}$$

Here $\gamma_1 = \langle \lambda_0^1, \delta_1 \rangle$ and $\gamma_2 = \langle \lambda_0^2, \delta_2 \rangle$ with $\delta_1(j_i) = \lambda_i^1$ and $\delta_2(j_i) = \lambda_i^2$ for all $j_i \in \{j_1 \ldots, j_m\}$. Inference rules, like productions, have premises and conclusions. However, the conclusion and premises of an inference rule may contain variables as placeholders for program fragments, and a rule may have side-conditions to restrict the possible instantiations of the variables. As such, an inference rule represents the (possibly infinite) set of productions containing those productions formed by each 'valid' combination of instantiations. An in-

stantiation is not valid if one or more of the side-conditions are not satisfied by it or if one of the transitions has a terminating source configuration. (We do not place restrictions on the instantiations of variables based on their names.) Side-conditions are written alongside premises rather than to the side of an inference rule. They are recognisable as those conditions that do not describe transitions. In what follows we use 'rule' for inference rules, not for productions. The following example rule has no premises and a single side-condition $X_1 \leqslant X_2$:

$$\frac{X_1 \leqslant X_2}{\{\mathit{leq}(X_1, X_2)\} \longrightarrow \{\mathbf{true}\}}$$

This rule defines the set of productions $\{\langle\langle\langle \mathit{leq}(z_1, z_2), \delta^0\rangle, \longrightarrow, \langle \mathbf{true}, \delta^0\rangle\rangle, \emptyset\rangle \mid z_1, z_2 \in \mathbb{Z}, z_1 \leqslant z_2\}$, if we assume that $\leqslant$ is only defined on elements of $\mathbb{Z}$ and that any term $\mathit{leq}(z_1, z_2)$ is not terminal with respect to $\longrightarrow$. Entity record δ^0 is the entity record with $\mathit{dom}(\delta^0) = \emptyset$.

Returning to the specification Φ for WHILE, the set of program fragments Λ is the union of all syntactic sets and auxiliary entity sets defined for WHILE in Chapter 6. The set of entity identifiers E is defined as $\{\mathtt{sto}, \mathtt{out}, \mathtt{sig}\}$. Two transition relations $\longrightarrow_E$, and $\longrightarrow_C$ are defined, together with their iterative closures $\dashrightarrow_E$ and $\dashrightarrow_C$, i.e. $R = \{\longrightarrow_E, \dashrightarrow_E, \longrightarrow_C, \dashrightarrow_C\}$. Literals are program terminals for expressions, and *done* is the single program terminal for commands, i.e. $\Lambda^T_{\longrightarrow_E} = L$ and $\Lambda^T_{\longrightarrow_C} = \{\mathit{done}\}$. An uncaught signal **cnt** results in termination. We specify that any configuration with $\mathtt{sig} = \mathbf{cnt}$ is terminal by defining $\Lambda^T_{\mathtt{sig}} = \{\mathbf{cnt}\}$. The other sets of program terminals are empty, i.e. $\Lambda^T_k = \emptyset$ for all $k \in \{\dashrightarrow_E, \dashrightarrow_E, \mathtt{sto}, \mathtt{out}\}$. The inference rules in Figures 7.1 and 7.2 define the productions of Φ. Commands evaluate any subexpressions in one transition as specified by the premises $\{E_1\} \dashrightarrow_E \{L_1\}$. An example derivation is given in Figure 7.3 on page 148.

Comparison with MSOS MRBS and MSOS rules achieve modularity with respect to auxiliary entities by allowing irrelevant entity references to be omitted from rules. In MSOS rules, any unreferenced entities are still represented, collectively, by label variables. Side-conditions on label variables limit the possible instantiations of label variables. In I-MSOS rules, label variables and side-conditions on label variables are only implicitly present. In MRBS productions, any unreferenced entities are not present in any way. It is when production rules come together as part of a derivation that any missing entities are added.

$$\frac{\{C_1\} \longrightarrow_C \{C_1'\}}{\{seq(C_1, C_2)\} \longrightarrow_C \{seq(C_1', C_2)\}}$$

$$\{seq(done, C_2)\} \longrightarrow_C \{C_2\}$$

$$\frac{\{E_1\} \dashrightarrow_E \{\mathbf{false}\}}{\{while(E_1, C_1)\} \longrightarrow_C \{done\}}$$

$$\frac{\{E_1\} \dashrightarrow_E \{\mathbf{true}\}}{\{while(E_1, C_1)\} \longrightarrow_C \{seq(listen\_cnt(C_1), while(E_1, C_1))\}}$$

$$\frac{\{E_1, \mathtt{sto} = \sigma_1\} \dashrightarrow_E \{L_1, \mathtt{sto} = \sigma_2\} \qquad \sigma_3 = \sigma_2[Id_1 \mapsto L_1]}{\{assign(Id_1, E_1), \mathtt{sto} = \sigma_1\} \longrightarrow_C \{done, \mathtt{sto} = \sigma_3\}}$$

$$\frac{\{E_1, \mathtt{out} = \alpha_1\} \dashrightarrow_E \{L_1, \mathtt{out} = \alpha_2\} \qquad \alpha_3 = \alpha_2 +\!\!+ [L_1]}{\{print(E_1), \mathtt{out} = \alpha_1\} \longrightarrow_C \{done, \mathtt{out} = \alpha_3\}}$$

$$\{continue, \mathtt{sig} = \mathbf{none}\} \longrightarrow_C \{done, \mathtt{sig} = \mathbf{cnt}\}$$

$$\frac{\{C_1, \mathtt{sig} = \mathbf{none}\} \longrightarrow_C \{C_1', \mathtt{sig} = \mathbf{none}\}}{\{listen\_cnt(C_1), \mathtt{sig} = \mathbf{none}\} \longrightarrow_C \{listen\_cnt(C_1'), \mathtt{sig} = \mathbf{none}\}}$$

$$\frac{\{C_1, \mathtt{sig} = \mathbf{none}\} \longrightarrow_C \{C_1', \mathtt{sig} = \mathbf{cnt}\}}{\{listen\_cnt(C_1), \mathtt{sig} = \mathbf{none}\} \longrightarrow_C \{done, \mathtt{sig} = \mathbf{none}\}}$$

$$\{listen\_cnt(done)\} \longrightarrow_C \{done\}$$

Figure 7.2: Rules for WHILE commands.

Restrictions on how the entities can be added are part of the definition of derivations. This definition follows the implicit propagation scheme suggested for I-MSOS and discussed in §6.3.3.

In MRBS, entities are part of configurations rather than part of a label. This makes it possible for entities to determine termination, as we have seen for the **cnt** signal in the WHILE specification. This is not directly possible in MSOS, as only configurations determine termination in an LTTS. However, it can be achieved by wrapping programs inside one or more auxiliary constructors that inspect entities and transition to values whenever a terminating entity is encountered. In his thesis, Bach Poulsen develops XSOS, a variant of MSOS

in which entities are part of configurations as well as of labels [Bach Poulsen, 2016]. A (non-)termination side-condition is added to rules to ensure that they do not define transitions over terminal configurations. This side-condition is implicitly present in Abbreviated XSOS rules, in a fashion similar to the implicit side-conditions of I-MSOS rules. In MRBS, there is no need for such a side-condition, as transitions are over non-terminating configurations.

In §6.2.2, it was suggested that the iterative closure of some relation $\longrightarrow$ captures the finite computations of the TTS for $\longrightarrow$. This is does not hold in the case of While, as there are computations in the TTS for $\longrightarrow_C$ that terminate because of a terminal entity $\mathtt{sig} = \mathbf{cnt}$, whereas there is no base case in the definition of $\dashrightarrow_C$ to capture this situation. For example, there is no derivation for the transition $\{seq(continue, done), \mathtt{sto} = \sigma, \mathtt{out} = \alpha, \mathtt{sig} = \mathbf{none}\} \dashrightarrow_C \{seq(done, done), \mathtt{sto} = \sigma, \mathtt{out} = \alpha, \mathtt{sig} = \mathbf{cnt}\}$, for all stores σ and output α. We cannot add the following rule, as $\{\lambda, \mathtt{sig} = \tau\}$ is terminal if $\tau \in \Lambda^T_{\mathtt{sig}}$.

$$\frac{\tau \in \Lambda^T_{\mathtt{sig}}}{\{\lambda, \mathtt{sig} = \tau\} \dashrightarrow_C \{\lambda, \mathtt{sig} = \tau\}}$$

We can, however, add the following base case for computations that involve at least one transition:

$$\frac{\{\lambda, \mathtt{sig} = \mathbf{none}\} \longrightarrow_C \{\lambda', \mathtt{sig} = \tau\} \qquad \tau \in \Lambda^T_{\mathtt{sig}}}{\{\lambda, \mathtt{sig} = \mathbf{none}\} \dashrightarrow_C \{\lambda', \mathtt{sig} = \tau\}}$$

If a computation terminates in a configuration with a non-terminating program component, the computation has terminated *abruptly*.

7.2 The Syntax of IML Programs

This section introduces the meta-language IML for developing MRBS specifications. IML programs are executable in the sense that each program defines an underlying interpreter for constructing derivations. The top-level transitions for which derivations are constructed are referred to as 'queries'. IML is intended as an intermediate meta-language; that is, specifications written in other meta-languages should be translatable into IML specifications. In Chapter 8 we employ IML as an intermediate language and give a translation from CBS

to IML.

This section defines the concrete and abstract syntax of IML programs. The core of an IML program is an IML specification consisting of relation declarations, entity declarations, terminal declarations, variable declarations and inference rule declarations. An IML program is an IML specification plus queries. Section 7.3 formalises the operational semantics of IML, showing how an interpreter is obtained from an IML program and showing how derivations are established for queries by calling the interpreter. The primary target is the function $interpreter_\phi$ that given a program fragment λ, an entity record δ and a relation symbol r returns a derivation $\langle\langle\lambda, \delta\rangle, r, \gamma_2\rangle$ or fails if there is none (according to Specification 1). We do not consider complete interpreters, computing all derivations, as we restrict ourselves to deterministic programming languages.

Whilst introducing the syntax of IML programs, this section describes informally how the declarations of an IML specification determine $interpreter_\phi$, starting with the following high-level summary.

Inference rules are generally read in a top-down fashion: if everything above the bar is true (the conditions) then we can conclude what is written below the bar (the conclusion). However, a derivation is typically established in a bottom-up fashion: beginning with the goal, we try to find a rule with a 'matching' conclusion such that all conditions are satisfied. Determining whether all conditions are satisfied involves finding sub-derivations for the premises. When there are multiple premises, we have to find multiple sub-derivations and the proof 'branches' as we have seen in Figure 7.3 on page 148. The process continues until all the branches are finalised by the successful application of an axiom. Whereas a human expert makes insightful decisions when forming the sub-derivations, software implementations often rely on some kind of brute force approach and use backtracking to undo incorrect decisions.

This process is automated for IML programs by considering each component of an inference rule as an 'action' that either finds variable instantiations or indicates that the rule is not applicable in the current context (the action fails). When a rule fails, an alternative rule is considered until one succeeds or all alternatives have been tried. To find variable instantiations efficiently, IML uses pattern matching, a well known procedure in functional programming and term rewriting [Peyton Jones, 1987, Visser, 2001]. We consider these aspects in more detail whilst developing the concrete and abstract syntax of IML programs in

Set name	Description
X	Set of all variables
E	Set of all entity identifiers
R	Set of all relation symbols
C	Set of all term constructor names
O	Set of all operation names
N	Set of all priority levels

Table 7.1: The basic syntactic sets of IML programs.

the style of §6.2.1.

The basic syntactic sets are listed in Table 7.1. The elements of the basic syntactic sets are to be determined by an implementation of IML, but are required to be pairwise disjoint.

7.2.1 Terms, Variables, Patterns and Expressions

IML uses a term-based representation of program fragments in which terms are constructed by applying term constructors (C) to zero or more terms. Terms may contain variables when they appear in inference rules and queries. Variables are placeholders for terms and it is the task of the interpreter to find their instantiations in order to form productions and derivations. Terms that contain variables are open terms, terms that do not contain variables are closed terms. The set of closed terms provided by an IML implementation determines the set Λ_Φ of program fragments in an MRBS specification Φ. Together, both kinds of terms are called symbolic terms and are captured by the derived syntactic set ST, defined by the grammar rules below[1]:

$$\begin{array}{lllll} ST & : \textit{ symbolic terms } & ::= & X & \{X_1\} \\ & & | & C\ (\ ST^{0,k}\ ) & \{\mathit{stapp}(C_1, ST_1, \ldots, ST_k)\} \end{array}$$

We use 'term' for symbolic term when it is not specified whether the term is open or closed. The set $CT \subset ST$ is the set of all closed terms.

The terms are are untyped; there are no constraints on the applicability of the constructors (constructors do not even have an arity). Although undesirable in principle, IML is intended as an intermediate-level meta-language and should therefore be liberal in order to support the specifications of various high-level meta-languages. Type-checking program

[1]The notation $X^{l,k}$ in a grammar rule indicates that a sequence of l or more comma-separated elements of X is expected and that these elements are available to the action of the rule as $X_1, \ldots, X_k$. If $X^{l\ k}$ is written instead of $X^{l,k}$, this indicates that whitespace-separated elements of X are expected.

fragments is considered to be an aspect of the static semantics of the high-level meta-language translating into IML.

Patterns have the same structure as symbolic terms but they are used differently. For this reasons we define patterns separately as the elements of the set P:

$$\begin{array}{lllll} P & : & \textit{patterns} & ::= & X & \{X_1\} \\ & & & | & C\ (\ P^{0,k}\ ) & \{\textit{papp}(C_1, P_1, \ldots, P_k)\} \end{array}$$

IML *expressions* are formed by terms or the application of an operation (O) to zero or more expressions. The primary purpose of expressions is to enable writing parts of side-conditions like $Z_1 \leqslant Z_2$ and $\alpha_1 +\!\!+ [L_1]$. These parts can be written as, for example, $\textit{lesser\_or\_equal}(Z_1, Z_2)$ and $\textit{list\_append}(\alpha_1, \textit{list\_singleton}(L_1))$, where Z_1, Z_2 and L_1 are placeholders for terms and *lesser_or_equal*, *list_append* and *list_singleton* are the names of operations. An implementation of IML determines the available operations and their semantics. A user of IML determines which operations to use and thus where the line is drawn between 'assumed semantics' and 'specified semantics'. The set of all expressions is defined by the grammar rule below:

$$\begin{array}{lllll} OE & : & \textit{operator expressions} & ::= & ST & \{ST_1\} \\ & & & | & O\ (\ OE^{0,k}\ ) & \{\textit{oeapp}(O_1, OE_1, \ldots, OE_k)\} \end{array}$$

Expressions and patterns occur in IML inference rules wherever we expect to find program fragments, but never in the same positions. Operationally, expressions are used to construct closed terms by instantiating variables and executing operations, whereas patterns are used to deconstruct closed terms by binding variables to sub-terms. This section shows where expressions appear and where patterns appear. Section 7.3 formalises the aforementioned operational processes.

7.2.2 Sequences and Sequence Variables

IML program fragments are sequences of terms, rather than single terms. In the concrete syntax, a sequence of expressions (patterns) is a comma-separated list of expressions (patterns), where the empty sequence is denoted by #. The sets *OES* and *PS* of all expression sequences and pattern sequences are defined by the following grammar rules:

$$
\begin{array}{rcllll}
OES & : & \textit{expression sequence} & ::= & \# & \{exprs()\} \\
 & & & | & OE^{1,k} & \{exprs(OE_1, \ldots, OE_k)\} \\
PS & : & \textit{pattern sequence} & ::= & \# & \{pats()\} \\
 & & & | & P^{1,k} & \{pats(P_1, \ldots, P_k)\}
\end{array}
$$

Variables in IML are placeholders for sequences of (closed) terms. Variables are given a *range* in variable declarations, providing a lower (and optional) upper bound on the length of the term sequence it binds. The set of all variable declarations is defined by the following grammar rule:

$$
\begin{array}{rcllll}
VDecl & : & \textit{variable declaration} & ::= & \textbf{seq-variable (}\ X\ \textbf{,}\ \mathbb{N}\ \textbf{)} & \{seq\_var(X_1, \mathbb{N}_1)\} \\
 & & & | & \textbf{seq-variable (}\ X\ \textbf{,}\ \mathbb{N}\ \textbf{,}\ \mathbb{N}\ \textbf{)} & \{seq\_var(X_1, \mathbb{N}_1, \mathbb{N}_2)\}
\end{array}
$$

7.2.3 Transitions

To define transitions we first define entity references and configurations. An entity reference is formed by an entity identifier and a sequence of expressions or an entity identifier and a sequence of patterns. The sets *ERef* and *PRef* are the sets of all *expression references* and *pattern references*:

$$
\begin{array}{rcllll}
ERef & : & \textit{expression references} & ::= & E = OES & \{eref(E_1, OES_1)\} \\
PRef & : & \textit{pattern references} & ::= & E = PS & \{pref(E_1, PS_1)\}
\end{array}
$$

An expression reference $E = OES$ determines that a given entity record is to be updated so that it maps the entity identifier E to the sequence of terms produced by evaluating the expressions OES. A pattern reference $E = PS$ determines that a given entity record is checked to confirm that it maps the entity identifier E to a sequence of terms that matches the pattern sequence PS.

Just like entity references, configurations are based either on expressions ($EConf$) or patterns ($PConf$). An *expression configuration* consists of a sequence of expressions and a sequence of expression references. A *pattern configuration* consists of a sequence of patterns and a sequence of pattern references.

$$
\begin{array}{rcll}
EConf & : & \textit{expression configurations} & \\
 & ::= & OES & \{econf(OES_1)\} \\
 & | & OES\ \textbf{,}\ ERef^{1,k} & \{econf(OES_1, ERef_1, \ldots, ERef_k)\} \\
PConf & : & \textit{pattern configurations} & \\
 & ::= & PS & \{pconf(PS_1)\} \\
 & | & PS\ \textbf{,}\ PRef^{1,k} & \{pconf(PS_1, PRef_1, \ldots, PRef_k)\}
\end{array}
$$

Note that, unlike in the previous section, configurations are not surrounded by braces.

A conclusion is a pattern configuration (source), a relation symbol, and an expression configuration (target). Conversely, a premise is an expression configuration (source), a relation symbol, and a pattern configuration (target). The sets CN and PR of all conclusions and premises respectively are defined by the grammar rules below:

$$\begin{array}{lcllllll} CN & : & \textit{conclusions} & ::= & PConf & R & EConf & \{concl(PConf_1, R_1, EConf_1)\} \\ PR & : & \textit{premises} & ::= & EConf & R & PConf & \{trans(EConf_1, R_1, PConf_1)\} \end{array}$$

This shows that IML transitions are directional. A premise is established by calling the underlying interpreter with the closed terms and entities produced by evaluating the expression configuration (source) as input. The closed terms and entities that the interpreter gives as output, if a transition is possible, are matched against the pattern configuration (target). A conclusion is treated the other way around. The source is a pattern configuration that should be matched by given closed terms and entities (interpreter input), whereas the target is an expression configuration that produces closed terms and entities (interpreter output) when evaluated.

7.2.4 Rules

Conditions IML rules have four different kinds of conditions: premises, termination conditions, nontermination conditions, and matching conditions. Premises have been discussed earlier. The three others are side-conditions that do not involve transitions. A termination condition is a pair of either an entity identifier and a sequence of expressions or a relation symbol and a sequence of expressions. A termination condition checks whether all the expressions in the sequence evaluate to terms which are considered terminal in the specification being defined (in §7.2.6 it is explained how the declarations of an IML specification determine sets of program terminals). A nontermination condition checks the opposite. A matching condition is a sequence of expressions and a sequence of patterns separated by $\triangleright$. The terms resulting from evaluating the expressions should match the pattern sequence for the condition to succeed. The set of all conditions is defined as CS by the following grammar rule (for convenience we define the set $ER = E \cup R$): :

$$
\begin{array}{lllll}
CS & : & conditions & ::= & PR & \{prem(PR_1)\} \\
 & & & | & OES \ \triangleright \ PS & \{pm(OES_1, PS_1)\} \\
 & & & | & \textbf{is-terminal} \ (\ ER\ ,\ OES\ ) & \{term(ER_1, OES_1)\} \\
 & & & | & \textbf{is-non-terminal} \ (\ ER\ ,\ OES\ ) & \{nterm(ER_1, OES_1)\}
\end{array}
$$

Rules An IML rule is a conclusion, a sequence of conditions, and a priority (element from N). The set *Rule* of all IML rules is defined by the following grammar rule:

$$
Rule \quad : \quad rules \quad ::= \quad \frac{CS^{0\ k}}{CN} \ (\ N\ ) \qquad \{rule(N_1, CS_1, \ldots, CS_k, CN_1)\}
$$

The conditions of IML rules are ordered, whereas the premises of MRBS productions are unordered. The relative order in which premises are executed matters because of implicit entity propagation. For example, if two premises add different value to a list of output, then the resulting output is dependent on the ordering of the premises. We assume that conditions are ordered early in the pipeline: in the original specification by the language specifier, by the translation from the high-level meta-language into IML, or by an IML preprocessor. An important property of the ordering is that each variable in a rule occurs first in a pattern before it occurs anywhere else, thus guaranteeing that all variables are instantiated when required. When checking this property, the source of a rule's conclusion is to be considered before any of the conditions, and the target of the conclusion is to be considered after any of the conditions. Alternatively, all valid orderings can be executed in a nondeterministic fashion, but we do not consider that option in this thesis.

An implementation may decide that the priority is optional in the concrete syntax by deciding on a default priority $N_1 \in N$.

7.2.5 Remarks on Determinism

In general it is difficult to determine whether a specification is deterministic, without running a complete interpreter on all possible input configurations. One problem is determining whether two or more rules are simultaneously applicable. Fortunately, it is often relatively easy for a human to check whether this is the case.

Nondeterministic specifications may be desirable, e.g. to define a transitive and reflexive transition relation (see §6.2.2), or to define a reflexive[2] big-step relation. This observation shows there is tension between two of the design targets for IML, which are to enable efficient

interpretation (on the one hand) for a wide range of specifications (on the other hand). We assume determinism and only fully execute the first rule that is found to be applicable. IML has a priority mechanism to enable users to guide the rule selection mechanism. We recommend a conservative use of priority levels in order to keep specifications easier to understand and verify. In particular, we think that mutually exclusive rules (rules that are never simultaneously applicable) should be given the same priority level when possible[3]. This keeps specifications simple, and enables implementations to order mutually exclusive rules based on alternative criteria. Priorities have other usages as well. For example, they can be used as an alternative to 'negative premises' [Groote, 1993] (an example is given in §8.1.2).

Another potential source of nondeterminism is that rules may have multiple possible variable instantiations. However, following from our design decisions, all IML conditions, including premises, are deterministic and result in at most one instantiation for each variable.

7.2.6 Specifications and Programs

Declarations and specifications An IML specification defines a rule-based specification Φ (Definition 7.1.5) with productions Ψ_Φ and a signature consisting of program fragments Λ_Φ, relation symbols $R_\Phi \subseteq R$, entity identifiers $E_\Phi \subseteq E$, and sets of program terminals $\{\Lambda^T_{\Phi,k}\}_{k\in(R_\Phi \cup E_\Phi)}$. The set of program fragments Λ_Φ is the set of all closed term sequences, i.e. $\Lambda_\Phi = CT^*$. The sets R_Φ, E_Φ, and $\{\Lambda^T_{\Phi,k}\}_{k\in(E\cup R)}$ are determined by relation declarations, entity declarations and termination declarations:

Decl	:	*declarations*	::=	*Rule*	$\{rule\_decl(Rule_1)\}$
			|	*VDecl*	$\{var\_decl(VDecl_1)\}$
			|	**relation** (R)	$\{rel\_decl(R_1)\}$
			|	**entity** (E , OES)	$\{ent\_decl(E_1, OES_1)\}$
			|	**terminal** (ER , C)	$\{term\_decl(ER_1, C_1)\}$

Relation declarations simply nominate elements from R for R_Φ. Entity declarations nominate elements from E for E_Φ, but are also associated with expression sequences. The expression sequence associated with entity e is evaluated to a program fragment which acts as the default for e whenever a query that does not refer to e is executed. A termination

[2] In the face of a reflexivity rule, this would cause nontermination in a way similar to how a recursive descent parser fails to terminate when acting on a left-recursive production.

[3] An approach in which priority is decided based on file-position is therefore not desired.

declaration is a pair of an element[4] from $ER = E \cup R$ and a constructor (element from C). A termination declaration $term\_decl(k, f)$ determines that a term constructed by f is terminal when it occurs in entity k or in the source of a transition over relation k. A non-empty sequence of terms is terminal with respect to k if all terms within it are. The empty sequence is not considered terminal. Let D_k be the set of terms which are terminal with respect to k, i.e. $stapp(f, t_0, \ldots, t_n) \in D_k$ if and only if k and f occur in a termination declaration $term\_decl(k, f)$. The set of program terminals for k is then defined as $\Lambda^T_{\Phi,k} = \{t_1, \ldots, t_m \mid t_1, \ldots, t_m \in \Lambda_\Phi, \forall_{1 \leqslant i \leqslant m}(t_i \in D_k), m \geqslant 1\}$.

As mentioned in §7.2.2, variable declarations restrict the length of the sequences the variable may bind. In practice it may be more convenient to associate variable declarations with rules so that the same variable may have different ranges in different rules. Our definitions do not prohibit implementations to do so; in such an implementation it is possible to rename the variables specific to a rule such that they differ from all other variables.

Queries and programs An IML specification is a sequence of order-independent declarations. An IML program is an IML specification and a sequence of queries. A query is syntactically identical to a premise:

Spec	:	*specifications*	::=	$Decl^{0\ k}$	$\{spec(Decl_1, \ldots, Decl_k)\}$
QRs	:	*queries*	::=	$PR^{0\ k}$	$\{queries(PR_1, \ldots, PR_k)\}$
Ω	:	*programs*	::=	*Spec QRs*	$\{program(Spec_1, QRs_1)\}$

In the grammar rule above we separate the declarations and the queries syntactically, which is done only for convenience. An implementation may choose to mix declarations and queries freely. As we shall see in §7.3.6, only the relative order of the queries matters semantically.

7.3 The Semantics of IML Programs

This section defines the operational semantic of IML programs by formalising a reference interpreter for IML given as the function *sem_program* at the very end of this section. The operational semantics of IML programs involves computing an interpreter for the IML specification embedded in each program and executing the program's queries by calling this

[4] In an implementation, a static check could ensure that the first component of a termination declaration is in fact an element from $E_\Phi \cup R_\Phi$.

interpreter. We present the semantics iteratively, first giving the operational semantics of pattern matching, operator expressions, conditions, rules, and finally queries. §7.3.2 explains how (closed) terms are matched against patterns. §7.3.3 shows how expressions evaluate to closed terms. §7.3.4 gives the semantics of conditions, forming the building blocks for the semantics of rules. §7.3.5 explains how an interpreter is obtained by backtracking between the rules of an IML specification. First we show how the declarations compute a signature.

7.3.1 Declarations

Section 7.2.6 explained how the declarations of an IML specification determine the signature of an MRBS specification Φ. In this subsection we show how the elements of the signature are computed from the declarations of an IML specification. The next subsections formalise how derivations are found based on the inference rules of an IML specification, leaving the productions Ψ_Φ that are used in this process implicit.

The set of program terminals is the set of closed term sequences, i.e. $\Lambda_\Phi = CT^*$.

Relation declarations nominate elements from R for R_Φ. The function *gather_rel* computes the set R_Φ, i.e. R_Φ is defined as $gather\_rel(\phi)$.

$$gather\_rel(spec(Decl_1, \ldots, Decl_k)) = gather\_rel(Decl_1, \ldots, Decl_k)$$
$$gather\_rel'(Decl_1, \ldots, Decl_k) =$$
$$\begin{cases} \emptyset & \textbf{if } k = 0 \\ \{R_1\} \cup gather\_rel'(Decl_2, \ldots, Decl_k) & \textbf{if } Decl_1 = rel\_decl(R_1) \\ gather\_rel'(Decl_2, \ldots, Decl_k) & \text{otherwise} \end{cases}$$

The set E_Φ is determined by the entity declarations. The function *gather_ent* computes an entity record from the entity declarations of a specification.

$$
\begin{aligned}
&\mathit{gather\_ent}(\mathit{spec}(\mathit{Decl}_1, \ldots, \mathit{Decl}_k)) = \mathit{gather\_ent}'(\delta_0)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) \\
&\quad \textbf{where } \delta_0(E') = \bot \\
&\mathit{gather\_ent}'(\delta_0)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) = \\
&\quad \begin{cases}
\delta_0 & \textbf{if } k = 0 \\
\mathit{gather\_ent}'(\mathit{mkRec}(\delta_0, E_1, \mathit{OE}_1))(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \textbf{if } \mathit{Decl}_1 = \mathit{ent\_decl}(E_1, \mathit{OE}_1) \\
& \quad \textbf{and } \varnothing \neq \mathit{eval\_exprs}(\mathit{OE}_1) \\
\bot & \textbf{if } \mathit{Decl}_1 = \mathit{ent\_decl}(E_1, \mathit{OE}_1) \\
& \quad \textbf{and } \varnothing = \mathit{eval\_exprs}(\mathit{OE}_1) \\
\mathit{gather\_ent}'(\delta_0)(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \text{otherwise}
\end{cases} \\
&\quad \textbf{where } \mathit{mkRec}(\delta_0, E_1, \mathit{OE}_1)(E') = \begin{cases} \mathit{eval\_exprs}(\mathit{OE}_1) & \textbf{if } E' = E_1 \\ \delta_0(E') & \text{otherwise} \end{cases}
\end{aligned}
$$

(The function *eval_exprs*, evaluating expressions to closed terms, is defined in §7.3.3. $\varnothing$ indicates evaluation failure.) For an IML specification ϕ, the set E_Φ is defined as $\mathit{dom}(\mathit{gather\_ent}(\phi))$. The entity record $\mathit{gather\_ent}(\phi)$ is thus complete by definition.

The sets $\Lambda^T_{\Phi,k}$, for each $k \in (E_\Phi \cup R_\Phi)$, are determined by a predicate *trm* computed from ϕ. The function *gather_trm* helps compute the predicate:

$$
\begin{aligned}
&\mathit{gather\_trm}(\mathit{spec}(\mathit{Decl}_1, \ldots, \mathit{Decl}_k)) = \mathit{gather\_trm}'(\mathit{trm}_0)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) \\
&\quad \textbf{where } \mathit{trm}_0(k)(t) = \textbf{false} \\
&\mathit{gather\_trm}(\mathit{trm}_0)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) = \\
&\quad \begin{cases}
\mathit{trm}_0 & \textbf{if } k = 0 \\
\mathit{gather\_trm}'(\mathit{mkTrm}(\mathit{ER}_1, C_1))(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \textbf{if } \mathit{Decl}_1 = \mathit{term\_decl}(\mathit{ER}_1, C_1) \\
\mathit{gather\_trm}'(\mathit{trm}_0)(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \text{otherwise}
\end{cases} \\
&\quad \textbf{where } \mathit{mkTrm}(\mathit{ER}_1, C_1)(k)(T_1) = \begin{cases} \textbf{true} & \textbf{if } k = \mathit{ER}_1 \textbf{ and } T_1 = \mathit{stapp}(C_1, \ldots) \\ \mathit{trm}_0(k)(T_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

A term sequence $t_1, \ldots, t_n$ is in Λ^T_k if and only if $n \geqslant 1$ and $\mathit{trm}(k, t_i)$ holds, for all $1 \leqslant i \leqslant n$.

Rather than showing how IML inference rules generate productions, we show how they generate derivations, leaving productions implicit. To find derivations, rules are considered in (descending) order of priority. We assume a total order $\leqslant_N$ over the priorities (elements of N) which, just like N, is determined by implementation. The relative ordering of rules with equal priority is arbitrary, but may greatly influence the efficiency (as shown in §13.2.1). An IML specification is deterministic when for all priorities it holds that all rules with that priority are mutually exclusive. Function *gather_rules*(r) finds all the rules of a specification

that define relation r. Function *order_rules* sorts the rules based on $\leqslant_N$. (We leave the sorting algorithm unspecified.)

$$
\begin{array}{l}
\mathit{gather\_rules}(r)(\mathit{spec}(\mathit{Decl}_1, \ldots, \mathit{Decl}_k)) = \mathit{order\_rules}(\mathit{gather\_rules}'(r)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k)) \\
\mathit{gather\_rules}'(r)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) = \\
\quad \begin{cases}
\epsilon & \textbf{if } k = 0 \\
\mathit{Rule}_1, \mathit{gather\_rules}'(r)(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \textbf{if } \mathit{Decl}_1 = \mathit{rule\_decl}(\mathit{Rule}_1) \\
 & \quad \textbf{and } \mathit{Rule}_1 = \mathit{rule}(\ldots, \mathit{concl}(p, r, e)) \\
\mathit{gather\_rules}'(r)(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \text{otherwise}
\end{cases} \\
\mathit{order\_rules}(R_0, \ldots, R_k) = \mathit{sortDescBy}(\leqslant_N)(R_0, \ldots, R_k) \\
\quad \textbf{where } \mathit{sortDescBy} = \ldots
\end{array}
$$

(Here ϵ denotes the empty sequence of rules.)

A variable declaration assigns a range to a variable, indicating the minimum and maximum number of closed terms the variable can bind. The minimum and maximum are any number greater or equal to zero. There may not be a maximum however, indicated with ∞. A mapping from variables to ranges is computed from a collection of variable declarations by *gather_range*:

$$
\begin{array}{l}
\mathit{gather\_range}(\mathit{spec}(\mathit{Decl}_1, \ldots, \mathit{Decl}_k)) = \mathit{gather\_range}'(\mathit{rng}_0)(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) \\
\quad \textbf{where } \mathit{rng}_0(x) = \langle 1, 1 \rangle \\
\mathit{gather\_range}'(\mathit{rng})(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) = \\
\quad \begin{cases}
\mathit{rng} & \textbf{if } k = 0 \\
\mathit{gather\_rules}'(\mathit{mkRange}(\mathit{rng})(\mathit{VD}_1))(\mathit{Decl}_2, \ldots, \mathit{Decl}_k) & \textbf{if } \mathit{Decl}_1 = \mathit{var\_decl}(\mathit{VD}_1) \\
\mathit{gather\_rules}'(\mathit{rng})(\mathit{Decl}_1, \ldots, \mathit{Decl}_k) & \text{otherwise}
\end{cases} \\
\quad \textbf{where } \mathit{mkRange}(\mathit{rng})(\mathit{VD}_1)(x) = \begin{cases}
\langle l, \infty \rangle & \textbf{if } \mathit{seq\_var}(x, l) = \mathit{VD}_1 \\
\langle l, r \rangle & \textbf{if } \mathit{seq\_var}(x, l, r) = \mathit{VD}_1 \\
\mathit{rng}(x) & \text{otherwise}
\end{cases}
\end{array}
$$

7.3.2 Pattern Matching and Substitution

Pattern matching destructs a given (closed) term with two purposes: determine whether it has a certain structure, and giving names to its constituents in the form of bindings. The former may be used to define lenient forms of equality (or similarity) and type-checking. The latter may be used to construct new terms out of the constituents and to analyse the constituents further. As a whole, pattern matching enables case analysis in inductive

definitions and proofs. Basic pattern matching is a simple yet powerful mechanism that underpins most if not all functional and declarative programming.

Pattern matching The outcome of pattern matching is an element of $\Theta \cup \{\varnothing\}$, where $\Theta = X \rightharpoonup CT^*$ is the set of all partial mappings from variables to closed term sequences and $\varnothing$ indicates failure. An element $\theta \in \Theta$ is referred to as a set of bindings, containing the binding $x_1 \mapsto \lambda_1$ if $\theta(x_1) = \lambda_1$. The outcomes of two matches are composed by the $\otimes$ operator, defined as follows:

$$\theta_1 \otimes \theta_2 = \begin{cases} \varnothing & \textbf{if } \varnothing = \theta_1 \\ \varnothing & \textbf{if } \varnothing = \theta_2 \\ \theta_1 \cdot \theta_2 & \text{otherwise} \end{cases}$$

Operator $\cdot$ is right-biased composition over sets of bindings. That is, when two patterns contain the same variable, the bindings produced by the right-most pattern (θ_2) are preferred. It can be checked statically whether two or more patterns contain the same variable.

Since IML constructors are variadic, we define pattern matching over sequences of closed terms and patterns. The sequences do not have to be of the same length as sequence-variables can bind sequences of length unequal to one. A pattern sequence with multiple sequence-variables is ambiguous in the sense that a term sequences may match a pattern sequence in multiple ways, resulting in different sets of bindings. To avoid ambiguity, we assume that pattern sequences have at most one sequence-variable. The functions *matches* defines pattern matching over sequences. The first argument of *matches* is a mapping from variables to ranges.

$$\begin{aligned} &\mathit{matches}(\mathit{rng}, T_1, \ldots, T_n, \mathit{pats}(P_1, \ldots, P_m)) = \\ &\quad \begin{cases} \theta_0 & \textbf{if } m = 0, n = 0 \\ \varnothing & \textbf{if } m = 0, n > 0 \\ \mathit{matches}(\mathit{rng}, T'_1, \ldots, T'_{n'}, \mathit{pats}(P'_1, \ldots, P'_{m'})) \otimes & \\ \quad \mathit{matches}(\mathit{rng}, T_2, \ldots, T_n, \mathit{pats}(P_2, \ldots, P_m)) & \textbf{if } P_1 = \mathit{papp}(C_2, P'_1, \ldots, P'_{m'}) \\ & \quad \textbf{and } T_1 = \mathit{stapp}(C_1, T'_1, \ldots, T'_{n'}) \\ & \quad \textbf{and } C_1 = C_2 \\ \mathit{match\_var}(\mathit{rng}, T_1, \ldots, T_n, \mathit{pats}(P_1, \ldots, P_m)) & \textbf{if } P_1 \in X \\ \varnothing & \text{otherwise} \end{cases} \\ &\quad \textbf{where } \theta_0(X_1) = \bot \end{aligned}$$

The functions *matches* recurses over the pattern sequences, deciding for each pattern which prefix of the term sequence matches (if any). If no patterns remain, pattern matching succeeds, but only if no terms remain. If the first pattern in the sequence (P_1) is a variable, function *match_var* is called to determine how many terms the variable should bind. This decision needs to respect the range of the variable (for which *rng* is used) and it needs to ensure that the rest of the term sequence can still match the rest of the pattern sequence.

$$match\_var(rng, T_1, \ldots, T_n, pats(X_1, P_1, \ldots, P_m)) = choose(l, min(n, r))$$
$$\textbf{where } \langle l, r\rangle = rng(X_1)$$
$$\textbf{and } min(n, r) = \begin{cases} n & \textbf{if } r = \infty \\ n & \textbf{if } n \leqslant r \\ r & \text{otherwise} \end{cases}$$
$$\textbf{and } choose(l, u) = \begin{cases} \varnothing & \textbf{if } u < l \\ [X_1 \mapsto T_1, \ldots, T_u] \otimes \theta_1 & \textbf{if } \theta_1 = matches(rng, T_{u+1}, \ldots, T_n, pats(P_1, \ldots, P_m)) \\ & \quad \textbf{and } \theta_1 \neq \varnothing \\ choose(l, u-1) & \text{otherwise} \end{cases}$$

A configuration is a pair $\langle \lambda_0, \delta_0\rangle$ with $\lambda_0 \in CT^*$ and $\delta_0 \in E \rightharpoonup CT^*$. The configuration is complete if δ_0 is defined on $E_\Phi \subset E$. We shall see that, by construction, all configurations propagated by interpreters generated for IML specifications are complete. The function *match_conf* formalises what it means for a complete configuration to match a pattern configuration (an element in *PConf*).

$$match\_conf(rng, \langle \lambda_0, \delta_0\rangle, pconf(PS_0, pref(E_1, PS_1), \ldots, pref(E_n, PS_n))) =$$
$$matches(rng, \lambda_0, PS_0) \otimes matches(rng, \delta_0(E_1), PS_1) \otimes \ldots \otimes matches(rng, \delta_0(E_n), PS_n)$$

Substitution Substitution is a process by which variables in a symbolic term are replaced by the closed terms they bind in a set of bindings. Substitution constructs terms, after pattern matching has taken terms apart and named their constituents. It is possible to check statically that all variables have bindings whenever they require substitution[5].

Substitution is formalised by the function *subs*. The function is partial, as an attempt to substitute an unbound variable is considered to be erroneous. The bases cases are $n = 0$

[5]Rule conditions may require reordering to ensure this. Finding such an order is not possible when conditions have cyclic dependencies.

and $S_0 \in X$. (The following definitions require that the application of a constructor to one or more undefined operands is undefined and that sequences concatenate.)

$$subs(\theta_1)(S_0) = \begin{cases} \perp & \textbf{if } S_0 \in X \textbf{ and } \perp = \theta_1(S_0) \\ \theta_1(S_0) & \textbf{if } S_0 \in X \\ stapp(C_1, subss(\theta_1, S_1 \ldots, S_n)) & \textbf{if } S_0 = stapp(C_1, S_1, \ldots, S_n) \end{cases}$$
$$subss(\theta_1)(S_1, \ldots, S_n) = subs(\theta_1)(S_1), \ldots, subs(\theta_1)(S_n)$$

Substitution can also be applied to expressions, yielding a closed expression (an expression consisting only of closed terms and closed expressions).

$$subs\_oe(\theta_1)(OE_0) = \begin{cases} subs(\theta_1)(OE_0) & \textbf{if } OE_0 \in ST \\ oeapp(O_1, subss\_oe(\theta_1)(OE_1, \ldots, OE_n)) & \\ \quad \textbf{if } OE_0 = oeapp(O_1, OE_1, \ldots, OE_n) & \end{cases}$$
$$subss\_oe(\theta_1)(OE_1, \ldots, OE_n) = subs\_oe(\theta_1)(OE_1), \ldots, subs\_oe(\theta_1)(OE_n)$$

To produce a configuration from an expression configuration, it is necessary to perform substitution on the expression configuration, to complete it using the defaults for unreferenced entities, and to evaluate the expressions within it. The first step is formalised by the function *subs_conf* defined below.

$$subs\_conf(\theta)(econf(OES_0, eref(E_1, OES_1), \ldots, eref(E_n, OES_n))) =$$
$$\quad econf(exprs(subss\_oe(\theta)(oes(OES_0))), eref(E_1, exprs(subss\_oe(\theta)(oes(OES_1)))), \ldots$$
$$\qquad , eref(E_n, exprs(subss\_oe(\theta)(oes(OES_n)))))$$
$$oes(exprs(OE_1, \ldots, OE_k)) = OE_1, \ldots, OE_k$$

7.3.3 Evaluating Expressions

To evaluate expressions, it is necessary to execute operations. For executing operations, we assume that an implementation of IML defines not only the set of available operations O, but also for each $O_1 \in O$ a function $operator_{O_1}$. When applied to a sequence of closed terms, $operator_{O_1}$ returns a sequence of closed term or indicates that it is not defined on the given terms. Evaluation fails ($\varnothing$) if an operation is applied to terms over which it is not defined or if one if its subexpressions fails to evaluate.

$$
\begin{aligned}
&\mathit{eval\_expr}(OE_0) = \\
&\quad \begin{cases} OE_0 & \textbf{if } OE_0 \in CT \\ \lambda_1 & \textbf{if } OE_0 = \mathit{oeapp}(O_1, OE_1, \ldots, OE_n) \\ & \quad \textbf{and } \lambda_1 = \mathit{operator}_{O_1}(\mathit{eval\_exprs}(OE_1, \ldots, OE_n)) \textbf{ and } \lambda_1 \neq \bot \\ \varnothing & \text{otherwise} \end{cases} \\
&\mathit{eval\_exprs}(OE_1, \ldots, OE_n) = \mathit{eval\_expr}(OE_1), \ldots, \mathit{eval\_expr}(OE_n)
\end{aligned}
$$

An expression configuration evaluates to a complete configuration by evaluating all (closed) expressions within it, as formalised by the function *eval_conf*. Any entities not referred to by the expression configuration are taken from a given entity record δ_0, which is assumed to be complete.

$$
\begin{aligned}
&\mathit{eval\_conf}(\delta_0)(\mathit{econf}(OES_0, \mathit{eref}(E_1, OES_1), \ldots, \mathit{eref}(E_n, OES_n))) = \\
&\quad \begin{cases} \varnothing & \textbf{if } \exists_{e \in dom(\delta_0)}(\delta_1(e) = \varnothing) \\ \langle \lambda_0, \delta_1 \rangle & \text{otherwise} \end{cases} \\
&\quad \textbf{where } \lambda_0 = \mathit{eval\_exprs}(\mathit{oes}(OES_0)) \\
&\quad \textbf{and} \qquad \delta_1(E') = \begin{cases} \mathit{eval\_exprs}(\mathit{oes}(OES_1)) & \textbf{if } E' = E_1 \\ \ldots & \ldots \\ \mathit{eval\_exprs}(\mathit{oes}(OES_n)) & \textbf{if } E' = E_n \\ \delta_0(E') & \textbf{if } E' \in dom(\delta_0) \setminus \{E_1, \ldots, E_n\} \end{cases}
\end{aligned}
$$

7.3.4 Conditions

All conditions have the potential to fail, indicating that the rule in which they occur is not applicable in the current context. The context is a complete entity record δ and a set of bindings θ. When executed successfully, a matching condition may extend the current set of bindings. Premises can both extend the current set of bindings as well as update the entity record. We define the set *State* as the set of all triples with bindings, entity records, and derivation sequences, i.e. $\mathit{State} = \Delta \times \Theta \times \mathcal{T}^*$, where Δ is $E_\Phi \to CT^*$ the set of all (complete) entity records, and $\mathcal{T}$ is the set of all (complete) derivations. The last component of *State* collects the (sub-)derivations found for the premises of a rule, in order to produce a derivation upon the successful execution of the rule. The semantics of a condition is given as a function that fails or returns a modified state given a state, i.e. the semantics of a condition is an element in $\mathit{State} \to (\{\varnothing\} \cup \mathit{State})$.

Matching conditions A matching condition $pm(OES_0, PS_0)$, concretely written as $OES_0 \rhd PS_0$, requires evaluating the closed expressions $subss(\theta)(oes(OES_0))$, where θ is the set of bindings in the current state, and matching the resulting term sequence to the pattern sequence PS_0.

$$sem\_pm_\phi(OES_0, PS_0)(\delta_0, \theta_0, \tau_0^*) =$$
$$\begin{cases} \langle \delta_0, \theta_0 \otimes \theta_1, \tau_0^* \rangle & \textbf{if } \theta_1 = matches(gather\_range(\phi), \lambda_1, PS_1) \textbf{ and } \theta_1 \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$
$$\textbf{where } \lambda_1 = eval\_exprs(subs\_oe(\theta_0)(oes(OE_0)))$$

(Variable τ_0^* (and later τ_1^*) is a placeholder for a sequence of derivations.)

Termination conditions Termination and nontermination conditions, concretely written as **is-terminal**(ER_1, OES_1) and **is-non-terminal**(ER_1, OES_1) respectively, are given an expression sequence OES_1 and an element $ER_1 \in E_\Phi \cup R_\Phi$. The expression sequence is evaluated to a closed term sequence λ_1 which is then checked for termination depending on ER_1 using the predicate $gather\_trm(\phi)(ER_1)$. The semantics of conditions $term(ER_1, OES_1)$ and $nterm(ER_1, OES_1)$ are given by the following functions:

$$sem\_term_\phi(ER_1, OES_1)(\delta_0, \theta_0, \tau_0^*) =$$
$$\begin{cases} \langle \delta_0, \theta_0, \tau_0^* \rangle & \textbf{if } is\_terminal_\phi(ER_1)(eval\_exprs(subss\_oe(\theta_0)(oes(OES_1)))) = \textbf{true} \\ \varnothing & \text{otherwise} \end{cases}$$
$$sem\_nterm_\phi(ER_1, OES_1)(\delta_0, \theta_0, \tau_0^*) =$$
$$\begin{cases} \langle \delta_0, \theta_0, \tau_0^* \rangle & \textbf{if } is\_terminal_\phi(ER_1)(eval\_exprs(subss\_oe(\theta_0)(oes(OES_1)))) = \textbf{false} \\ \varnothing & \text{otherwise} \end{cases}$$
$$is\_terminal_\phi(ER_1)(T_1, \ldots, T_n) = \begin{cases} \textbf{false} & \textbf{if } n = 0 \\ \textbf{true} & \textbf{if } \forall_{1 \leqslant i \leqslant n} gather\_trm(\phi)(ER_1)(T_i) = \textbf{true} \\ \textbf{false} & \text{otherwise} \end{cases}$$

Both conditions fail if the expression sequence is not evaluated successfully.

Premises To give the semantics of premises we rely on the function $interpreter_\phi$ that, given a complete configuration γ_1 and a relation symbol r, returns a derivation τ with root $\langle \gamma_1, r, \gamma_2 \rangle$, for some complete configuration γ_2, or $\varnothing$ if there is no such derivation. In the former case, the third component of the *State* is extended with τ. The definition of

$interpreter_\phi$ is given in §7.3.5.

$$sem\_prem_\phi(e,r,p)(\delta_0,\theta_0,\tau_0,\ldots,\tau_k) =$$
$$\begin{cases} \langle \delta_2, \theta_0 \otimes \theta_1, \tau_0, \ldots, \tau_k, \tau \rangle & \textbf{if } \langle \lambda_1, \delta_1 \rangle = eval\_conf(\delta_0)(subs\_conf(\theta_0)(e)) \\ & \textbf{and } \tau = interpreter_\phi(\langle \lambda_1, \delta_1 \rangle, r) \textbf{ and } \tau \neq \varnothing \\ & \textbf{and } \langle \lambda_2, \delta_2 \rangle = tgt(root(\tau)) \\ & \textbf{and } \theta_1 = match\_conf(\langle \lambda_2, \delta_2 \rangle, p) \textbf{ and } \theta_1 \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

A premise fails if evaluating the expression configuration fails, if the call to the interpreter fails or if matching the resulting configuration against the pattern configuration fails.

The helper function *sem_cond* selects the right semantic function for a given condition:

$$sem\_cond_\phi(CS_0) = \begin{cases} sem\_prem_\phi(e,r,p) & \textbf{if } CS_0 = trans(e,r,p) \\ sem\_term_\phi(ER_0, OES_0) & \textbf{if } CS_0 = term(ER_0, OES_0) \\ sem\_nterm_\phi(ER_0, OES_0) & \textbf{if } CS_0 = nterm(ER_0, OES_0) \\ sem\_pm_\phi(OES_0, PS_0) & \textbf{if } CS_0 = pm(OES_0, PS_0) \end{cases}$$

A condition is a function from an input state to an output state whereas an interpreter is effectively a function from an input configuration γ_1 to an output configuration γ_2 (both may fail) proving that there is a transition from γ_1 to γ_2 according to some relation r, also given as input, by means of a derivation with root $\langle \gamma_1, r, \gamma_2 \rangle$. A premise is a special condition in the sense that it requires the construction of a derivation by calling an interpreter with an input configuration constructed by evaluating the expression configuration of the premise.

7.3.5 Rules and Interpreters

The possibly many conditions of an IML rule are considered in the order that they are given. The order matters, as some conditions modify the given state (e.g. the state's entity record) whilst the behaviour of conditions is also state-dependent. The conditions of a rule are therefore considered as a sequence in which state is propagated, as specified by *sem_seq* below. A sequence of conditions fails as soon as one condition fails.

$$sem\_seq(c_0,\ldots,c_k)(\delta_0,\theta_0,\tau_0^*) =$$
$$\begin{cases} \langle \delta_0, \theta_0, \tau_0^* \rangle & \textbf{if } k = 0 \\ \varnothing & \textbf{if } \varnothing = c_0(\delta_0, \theta_0, \tau_0^*) \\ sem\_seq(c_1,\ldots,c_k)(\delta_1,\theta_1,\tau_1^*) & \textbf{if } \langle \delta_1, \theta_1, \tau_1^* \rangle = c_0(\delta_0, \theta_0, \tau_0^*) \end{cases}$$

Function *sem_lhs* expresses the interpretation of pattern configurations as if they are conditions (for convenience). The function matches a given closed configuration $\langle \lambda_0, \delta_0 \rangle$ to the pattern configuration of the rule's conclusion (p below).

$$sem\_lhs(\lambda_0)(p)(\delta_0, \theta_0, \tau_0^*) = \begin{cases} \varnothing & \textbf{if } \varnothing = match\_conf(\langle \lambda_0, \delta_0 \rangle, p) \\ \langle \delta_0, \theta_0 \otimes \theta_1, \tau_0^* \rangle & \textbf{if } \theta_1 = match\_conf(\langle \lambda_0, \delta_0 \rangle, p) \end{cases}$$

Given a closed configuration $\gamma_0 = \langle \lambda_0, \delta_0 \rangle$, a rule executes each of its conditions and, if successful, evaluates the target of its conclusion (an expression configuration) to yield a closed configuration γ_1. The result of a executing a rule is either failure or a derivation with root $\langle \gamma_0, r, \gamma_1 \rangle$, whose children are the derivations found for each of the premises (in the order the premises were executed). The relation symbol r is that of the rule's conclusion.

$$sem\_rule_\phi(rule(N_1, CS_0, \ldots, CS_k, concl(p, r, e)))(\lambda_0, \delta_0) =$$
$$\begin{cases} \varnothing & \textbf{if } \varnothing = b(\delta_0, \theta_0, \epsilon) \\ \langle \langle \langle \lambda_0, \delta_0 \rangle, r, \gamma_1 \rangle, \tau_0, \ldots, \tau_m \rangle & \textbf{if } \langle \delta_1, \theta_1, \tau_0, \ldots, \tau_m \rangle = b(\delta_0, \theta_0, \epsilon) \\ & \quad \textbf{and } \gamma_1 = eval\_conf(\delta_1)(subs\_conf(\theta_1)(e)) \end{cases}$$
$$\textbf{where } b = sem\_seq(sem\_lhs(\lambda_0)(p), sem\_cond_\phi(CS_0), \ldots, sem\_cond_\phi(CS_k))$$
$$\textbf{and} \quad \theta_0(X_1) = \bot$$

(Here ϵ denotes the empty sequence of derivations.)

Backtracking selects between rules until the first applicable rule is found, as specified by *sem_branches*:

$$sem\_branches_\phi(R_0, \ldots, R_k)(\lambda_0, \delta_0) =$$
$$\begin{cases} \varnothing & \textbf{if } k = 0 \\ sem\_branches_\phi(R_1, \ldots, R_k)(\lambda_0, \delta_0) & \textbf{if } \varnothing = sem\_rule_\phi(R_0)(\lambda_0, \delta_0) \\ sem\_rule_\phi(R_0)(\lambda_0, \theta_0) & \text{otherwise} \end{cases}$$

The interpreter for a specification ϕ is defined in terms of $sem\_branches_\phi$, *gather_rules* and $is\_terminal_\phi$ (there is no derivation if the input configuration $\langle \lambda_0, \delta_0 \rangle$ is terminal):

$$interpreter_\phi(\langle \lambda_0, \delta_0 \rangle, r) =$$
$$\begin{cases} \varnothing & \textbf{if } is\_terminal_\phi(r)(\lambda_0) \\ \varnothing & \textbf{if } \exists_{j \in dom(\delta_0)}(is\_terminal_\phi(j)(\delta_0(j))) \\ sem\_branches(R_0, \ldots, R_k)(\lambda_0, \delta_0) & \text{otherwise} \end{cases}$$
$$\textbf{where } R_0, \ldots, R_k = gather\_rules(r)(\phi)$$

7.3.6 Queries and Programs

A query is a premise that is executed outside a rule, to form a derivation for the transition described by the query. A query is executed from a default state, which contains the entity record obtained from the entity declarations by *gather_ent*. The result of a query is failure or a derivation. The former suggests[6] that the transition described by the query does not hold. The latter shows that it does hold, and gives a derivation that proves it. A set of bindings θ is propagated between queries, and the expression configuration on the left-hand side of a query need not be closed. This makes it possible to incrementally build queries by referring to the outcomes of previous queries.

$$sem\_query_\phi(trans(e,r,p))(\theta) = \begin{cases} \varnothing & \textbf{if } \varnothing = sem\_prem_\phi(e,r,p)(\delta_0,\theta,\epsilon) \\ \langle \tau_1, \theta_1 \rangle & \textbf{if } \langle \delta_1, \theta_1, \tau_1 \rangle = sem\_prem_\phi(e,r,p)(\delta_0,\theta_0,\epsilon) \end{cases}$$
$$\textbf{where } \delta_0 = gather\_ent(\phi)$$

A sequence of queries accumulates any discovered bindings, similar to a sequence of conditions. Unlike a sequence of conditions, entity records are not propagated between queries and if a query fails, any remaining queries are still executed.

$$sem\_queries_\phi(Q_0,\ldots,Q_k)(\theta_0) =$$
$$\begin{cases} \theta_0 & \textbf{if } k = 0 \\ sem\_queries_\phi(Q_1,\ldots,Q_k)(\theta_1) & \textbf{if } \langle \theta_1, \tau_1 \rangle = sem\_query_\phi(Q_0)(\theta_0) \\ sem\_queries_\phi(Q_1,\ldots,Q_k)(\theta_0) & \textbf{if } \varnothing = sem\_query_\phi(Q_0)(\theta_0) \end{cases}$$

An implementation could output the derivation τ produced by each successful query and indicate that a query has failed otherwise.

Finally, the semantics of an IML program is expressed by the function *sem_program*.

$$sem\_program(program(\phi, queries(Q_0,\ldots,Q_k))) = sem\_queries_\phi(Q_0,\ldots,Q_k)(\theta_0)$$
$$\textbf{where } \theta_0(X_1) = \bot$$

7.3.7 Remarks on Completeness

Executing a premise results in at most one new state, following the assumption that specifications are deterministic. If an IML specification is nondeterministic, then premises and

[6] As discussed in §7.3.7, there are reasons why a derivation may not have been discovered.

queries may fail even though a derivation for the premise or query exists in the underlying MRBS specification (false negatives). If a premise is executed as part of a rule r, and the premise can be established in two ways, resulting in states s_1 and s_2, then it may be the case that any subsequent conditions of r fail in state s_1 but succeed in state s_2. By considering only one of s_1 and s_2, a false negative is encountered if s_1 happens to be chosen.

False negatives may also arise from the order in which premises are executed. If a rule has two premises p_1 and p_2, the premises may be such that executing p_1 yields entity values under which p_2 cannot be established, whereas p_2 does not have this effect on p_1. A sequence of conditions is only considered in one order, and if p_1 happens to be considered before p_2, the result is a false negative. This is not an issue in specifications in which rules have at most one premise, which is typical for small-step specifications.

7.4 An Implementation of IML

The previous sections have presented IML in a general fashion, ignoring certain practical considerations. For example, we have not determined priority levels nor their relative ordering. In this section we discuss some design decisions made during the development of the IML interpreter (package `iml-tools`). At the core is a more or less direct Haskell implementation of the operational semantics described in Section 7.3. The implementation piggy-backs on the Haskell Funcon Framework by using its value operations.

7.4.1 Values and Operations

In the previous sections we have ignored that term representations are not suitable for representing certain types of values. For example, it is impractical to represent integers with constructors or constructor applications. Moreover, it is impossible to represent the unorderedness and uniqueness properties of set elements structurally. For this reason, CBS has a collection of 'built-in' values, types, and operations on values [Van Binsbergen et al., 2016], some of which have axiomatic definitions or are specified informally. The built-ins of CBS have been implemented in the Haskell package `funcons-values` and are exported by the module `Funcons.Operations`. Some built-in values are native to Haskell, such as floating-point numbers and ASCII characters. Other types of values are provided by libraries, such

as vectors and multisets[7]. Most of the value operations are directly implemented by a function provided by an existing Haskell library. In other cases, the operation has been easy to define in terms of existing operations or functions.

We built an interpreter for IML on top of the `funcons-values` package so that the IML interpreter has the same built-in values and operations as CBS. In what follows, 'values' refers to elements of the *Values* datatype of `funcons-values`, 'types' to elements of *Types*, and 'term' to elements of the *Term* datatype of the IML interpreter. Types are values, and values are terms[8].

The IML interpreter has a keyword for each built-in type, e.g. **integers** and **sets**. These keywords are used to indicate that values of that type are terminal with respect to a particular transition relation $\longrightarrow$. For example:

$$\mathbf{terminal}(\longrightarrow, \mathbf{integers})$$
$$\mathbf{terminal}(\longrightarrow, \mathbf{sets})$$

Types are values of type **types**. All values are values of type **values**. It is thus possible to indicate that all values are terminal with respect to $\longrightarrow$ with a single declaration:

$$\mathbf{terminal}(\longrightarrow, \mathbf{values})$$

Types, values, and terms can be formed by applying a constructor to zero or more terms. 'Value constructors' and 'type constructors' are for implementing user-defined types. It can be helpful to give term constructors and value constructors the same name. In an implementation, however, the two need to be distinguishable. For this reason, our IML implementation has three forms of application: $f(x_1, \ldots, x_n)$ denotes a term formed by applying term constructor f, $f_v\langle x_1, \ldots, x_n\rangle$ denotes a value formed by applying value constructor f_v, and $f_t[x_1, \ldots, x_n]$ denotes a type formed by applying type constructor f_t. To distinguish occurrences of nullary constructors we simply write $f()$, $f\langle\rangle$, or $f[]$ for any term, value, or type constructor.

[7] The `vector` and `multiset` packages on Hackage.
[8] An injection function applies a 'wrapper' constructor, a projection function removes it if present.

7.4.2 Priority Levels

Priorities between rules can be implemented in many ways. Here we make a suggestion based on a number of observations. Firstly, the priority levels should form a total order. Secondly, it should be easy to keep track of a specification's priority levels and their relative ordering. Thirdly, it should be possible to introduce a level in between two existing levels, without having to change any rules.

Consider choosing the set $\mathbb{N}$ for priority levels N (Table 7.1), and the situation in which a rule r_1 has priority 1 and a rule r_2 has priority 2. If a new rule r_1' is preferred over r_1, but r_2 over r_1', then it is necessary to 'bump' the priority level of r_2 to a number larger than 2, in order to give r_1' priority 2. This problem is overcome with floating-point numbers, as r_1' can then be given priority 1.5 (assuming r_1 had 1.0 and r_2 had 2.0).

To make it easier to keep track of priority levels, we introduce priority level declarations that associate mnemonic names with priority levels. For example:

$$\textbf{priority}(\mathit{DEFAULT}, 5.0)$$
$$\textbf{priority}(\mathit{FALLBACK}, 1.0)$$
$$\textbf{priority}(\mathit{OVERRIDE}, 6.0)$$

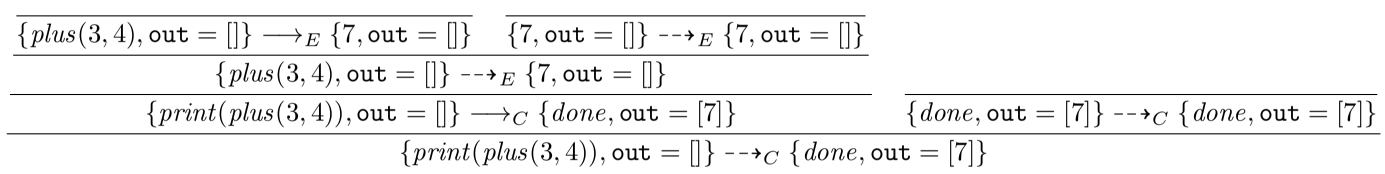


Figure 7.3: Derivation of $\{print(plus(3,4)), \texttt{out} = []\} \dashrightarrow_C \{done, \texttt{out} = [7]\}$. The derivation omits the entities `sto` and `sig`.

Chapter 8

CBS to IML Translation

Acknowledgements *The CBS language and the funcons definitions given in Sections 8.1, 8.2, 8.3, 8.4, and 8.5 are authored by Peter Mosses and Neil Sculthorpe.*

The PLANCOMPS project has developed a component-based approach to formal semantics. One of the outcomes of the project is a formal specification language called Component-Based Semantics (CBS). At the heart of CBS is a library of highly reusable fundamental constructs (funcons, for short), which is provided alongside CBS and can be used across specifications. The funcons are defined in CBS themselves and are not altered after their release, thereby fixing language specifications that depend on them. CBS specifications refer to funcons unambiguously, without the need for namespaces or version control.

A CBS specification consists of three parts: abstract syntax definitions, funcon definitions, and translation functions. Translation functions give a denotational semantics to the specified language, translating abstract syntax trees to funcon terms. The funcons themselves have an operational semantics, defined via inference rules. A CBS specification is intended to be executable so that languages can be tested whilst being developed. Parsers can be generated from the syntax definitions, if adequate disambiguation strategies have been identified. The syntax trees produced by the parsers are translated to funcon terms by applying the translation functions. Finally, funcon terms are executed by running an interpreter generated from the funcon definitions. In this thesis we focus on the funcon definitions of CBS and describe two methods for executing funcon terms, in this and the

next chapter.

The beta release of CBS is accompanied by example specifications of deterministic programming languages. Funcons capturing the fundamental constructs of concurrent languages are not part of the beta release and are currently being developed. An example of how MSOS can be used to specify concurrency constructs is given by [Mosses, 1999].

By giving examples, this chapter introduces funcons and shows how funcon definitions are translated to IML. The translation has been implemented in a compiler for CBS (package `funcons-intgen`) and has been used to generate composable IML specifications for a majority of the funcons in beta-release. The translation has been tested by executing tests as IML queries. This chapter expects that an intuitive understanding of the behaviour of the used funcons can be deduced from their names or their online documentation available at `http://cbsbeta.ltvanbinsbergen.nl`.

8.1 Preliminaries

Funcon definitions consist of a signature and a (possibly empty) collection of CBS inference rules. CBS inference rules have their foundations in I-MSOS (§6.3.3) and involve three relations that capture computational steps ($\longrightarrow$), context-insensitive rewriting ($\rightsquigarrow$), and dynamic typing (:). Relations $\longrightarrow$ and $\rightsquigarrow$ describe how funcon terms evaluate, following the theory on value-computation transition systems (VCTS) [Churchill and Mosses, 2013]. The operational details of $\rightsquigarrow$ are discussed in §8.1.1. The dynamic typing relation is used for case analysis in rules.

A CBS rule is referred to as a computation rule if it has a $\longrightarrow$-transition as its conclusion, or is referred to as a rewrite rule otherwise, in which case it has a $\rightsquigarrow$-transition as its conclusion. Thus, the $\longrightarrow$-relation is defined by computation rules, and the $\rightsquigarrow$-relation by rewrite rules. Some computation rules are implicitly determined by the signatures of funcons (see §8.1.4). Some rewrite rules are implicit as well, as discussed in Section 8.3. The typing relation is implicitly defined by type synonyms and datatype definitions. All rules become explicit in the generated IML. Section 8.2 shows the IML rules generated from type definitions. Sections 8.3 and 8.4 show how IML rules are generated from rewrite rules and computation rules respectively. This section gives the necessary IML declarations

seq-variable$(Xs, 0)$	
seq-variable$(Xs', 0)$	**priority**(LOWER, 3.0)
seq-variable$(Xs'', 0)$	**priority**(MEDIUM, 5.0)
seq-variable$(Xp, 1)$	**priority**(HIGHER, 7.0)
seq-variable$(Xp', 1)$	**terminal**$(\longrightarrow, \textbf{values})$
seq-variable$(Vs, 0)$	**terminal**$(\rightsquigarrow, \textbf{values})$
seq-variable$(Ts, 0)$	

Figure 8.1: IML declarations active throughout this chapter.

independent of any CBS input.

Figure 8.1 lists sequence variables declaration, **terminal**-declarations, and priority levels. The **terminal**-declarations show that all built-in values are terminal with respect to the $\longrightarrow$ and $\rightsquigarrow$ relations.

To distinguish IML constructors and funcons, we type-set funcons in **this** way. This chapter relies on some of the specifics of our IML implementation discussed in Section 7.4.

8.1.1 Rewrites

The relations $\longrightarrow$ and $\rightsquigarrow$ are the transition relation and rewriting relation of a value-computation transition system (VCTS) [Churchill and Mosses, 2013]. The rewrite relation of a VCTS is reflexive and transitive, and rewriting can happen before and after every $\longrightarrow$-transition (called 'saturation' in [Churchill and Mosses, 2013]). Moreover, the relation is a precongruence with respect to the constructors of the VCTS. Rules for reflexivity, transitivity, saturation and precongruence are implicitly present in CBS specifications. These rules are highly non-deterministic and we do not define IML variants of these rules. Instead we 'inline' applications of these rules where necessary. For example, rather than giving a rule for saturation, we perform as many rewrites as possible to any term which appears on the left-hand side of a premise. Instead of adding reflexivity and transitivity rules for $\rightsquigarrow$, we define a relation $\rightsquigarrow^*$ such that $t_0 \rightsquigarrow^* t_n$ if there is a finite sequence $t_0 \ldots t_n$ with $t_i \rightsquigarrow t_{i+1}$, for all $0 \leqslant i < n$ (recall that # denotes the empty sequence):

$$\frac{}{X \rightsquigarrow^* X}\ \text{(LOWER)} \qquad (8.1)$$

$$\frac{X \rightsquigarrow Xs \quad Xs \rightsquigarrow^* Xs'}{X \rightsquigarrow^* Xs'}\ \text{(MEDIUM)} \qquad (8.2)$$

$$\frac{}{\# \rightsquigarrow^* \#}\ \text{(MEDIUM)} \qquad (8.3)$$

$$\frac{X \rightsquigarrow^* Xs \quad Xp \rightsquigarrow^* Xs'}{X, Xp \rightsquigarrow^* Xs, Xs'}\ \text{(MEDIUM)} \qquad (8.4)$$

Note that the definition of $\rightsquigarrow^*$ is very similar to how the iterative closure (see §6.2.2) of $\rightsquigarrow$ would be defined, with the difference that there is no termination requirement (terms do not need to rewrite to values). It follows from the priority levels of Rules (8.1) and (8.2) that $\rightsquigarrow^*$ rewrites as much as possible ($t_0 \ldots t_n$ is the longest finite sequence with the aforementioned property). Rewrites can be performed over a sequence of terms (Rules (8.3) and (8.4)) as well as over single terms. As shown in the later sections of this chapter, to account for saturation, whenever there is a premise $X \longrightarrow Y$ in a CBS rule, the corresponding IML rule has the premises $[\![X]\!] \rightsquigarrow^* X'$ and $X' \longrightarrow [\![Y]\!]$, where $[\![X]\!]$ and $[\![Y]\!]$ are the translations of X and Y, and X' is some fresh variable.

8.1.2 Dynamic Typing

Relation $\longrightarrow$ is treated like a function f, in the sense that the transition $X \longrightarrow Y$ holds if the output of f is Y given input X. The $:$ relation is treated differently. In the case of $V : T$, we consider both V and T as input. In the IML specification we define a relation $\Rightarrow_{ty}$, whose left-hand side is a sequence of values, the first of which a type. The right-hand side is a Boolean (either $true\langle\rangle$ or $false\langle\rangle$). A transition $T, V \Rightarrow_{ty} true\langle\rangle$ indicates that V is of type T.

The following rule is a fall-back rule; it has the lowest priority and is only executed if no other rules are applicable:

$$\frac{}{T, Xs \Rightarrow_{ty} false\langle\rangle}\ \text{(LOWER)} \qquad (8.5)$$

Because of this rule, there is a $\Rightarrow_{ty}$-transition for any non-empty sequence.

The built-in operation `type-member` exists to type-check built-in values against built-in types. It is given two arguments, an arbitrary value V and a type T, and returns whether V

is of type T. The following rule determines $T, V \Rightarrow_{ty} \mathit{true}\langle\rangle$ if V is a member of T according to `type-member`:

$$\frac{\texttt{type-member}(V, T) \rhd \textbf{true}\langle\rangle}{T, V \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.6}$$

Types may be the result of rewriting terms. For example, the term *values* rewrites to the result of applying the operation `values` to zero arguments.

$$\frac{}{\mathit{values} \rightsquigarrow \texttt{values}}\ \text{(MEDIUM)} \tag{8.7}$$

The operation `values` returns the internal representation of the built-in type **values** (all values are of types **values**).

User-defined types also require rewrites, as explained in Section 8.2. We therefore introduce the relation $\overset{\rightsquigarrow}{\Rightarrow}_{ty}$, for rewriting before type-checking:

$$\frac{\mathit{Xp} \rightsquigarrow^{*} \mathit{Xp}' \quad \textbf{is-terminal}(\rightsquigarrow, \mathit{Xp}') \quad \mathit{Xp}' \Rightarrow_{ty} B}{\mathit{Xp} \overset{\rightsquigarrow}{\Rightarrow}_{ty} B}\ \text{(MEDIUM)} \tag{8.8}$$

Type operators CBS supports a number of type operators such as | for type-union, & for type-intersection, and ∼ for type-complement (type-negation). To represent these operators as terms, we introduce the type constructors *tyunion*, *tyinter*, and *tyneg* respectively, together with the following typing rules:

$$\frac{T_1, X \Rightarrow_{ty} \mathit{true}\langle\rangle}{\mathit{tyunion}[T_1, T_2], X \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.9}$$

$$\frac{T_2, X \Rightarrow_{ty} \mathit{true}\langle\rangle}{\mathit{tyunion}[T_1, T_2], X \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.10}$$

$$\frac{T_1, X \Rightarrow_{ty} \mathit{true}\langle\rangle \qquad T_2, X \Rightarrow_{ty} \mathit{true}\langle\rangle}{\mathit{tyinter}[T_1, T_2], X \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.11}$$

$$\frac{T, X \Rightarrow_{ty} \mathit{false}\langle\rangle}{\mathit{tyneg}[T], X \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.12}$$

Other type operators construct types for type-checking value sequences, e.g. **integers*** is the type of integer sequences, **booleans**$^{?}$ is the type of optional Booleans (empty or

singleton Boolean sequences), and **values**$^+$ is the type of non-empty value sequences. To represent these operators, we introduce the type constructors *tystar*, *tyopt*, and *typlus* with the following rules:

$$\frac{}{tystar[T] \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.13}$$

$$\frac{T, X \Rightarrow_{ty} true\langle\rangle \quad tystar[T], Xs \Rightarrow_{ty} true\langle\rangle}{tystar[T], X, Xs \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.14}$$

$$\frac{T, X \Rightarrow_{ty} true\langle\rangle}{typlus[T], X \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.15}$$

$$\frac{T, X \Rightarrow_{ty} true\langle\rangle \quad typlus[T], Xs \Rightarrow_{ty} true\langle\rangle}{typlus[T], X, Xs \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.16}$$

$$\frac{}{tyopt[T] \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.17}$$

$$\frac{T, X \Rightarrow_{ty} true\langle\rangle}{tyopt[T], X \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.18}$$

In CBS, a sequence of types is written within parenthesis. For example the type of sequences containing an integer and a Boolean is written as (**integers**, **booleans**). We introduce the constructor *tyseq* for wrapping type sequences, together with the following rules:

$$\frac{}{tyseq[] \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.19}$$

$$\frac{T, X \Rightarrow_{ty} true\langle\rangle \quad tyseq[Ts], Xs \Rightarrow_{ty} true\langle\rangle}{tyseq[T, Ts], X, Xs \Rightarrow_{ty} true\langle\rangle}\ (\texttt{MEDIUM}) \tag{8.20}$$

The $\wedge$ operator is used to construct type sequences of a particular type repeated n times. For example, **integers**$^{\wedge 3}$ = (**integers**, **integers**, **integers**). We introduce the constructor *typower*, whose second argument is a natural number — not a type — and the following

rules:

$$\frac{T, X \Rightarrow_{ty} \mathit{true}\langle\rangle \quad \texttt{integer-subtract}(N, 1) \rhd N' \quad \mathit{typower}[T, N'], \mathit{Xs} \Rightarrow_{ty} \mathit{true}\langle\rangle}{\mathit{typower}[T, N], X, \mathit{Xs} \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.21}$$

$$\frac{}{\mathit{typower}[T, 0] \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.22}$$

The operands of type operators may require evaluation. We therefore introduce a term constructor for each type constructor corresponding to a type operator. The term constructors have rules to evaluate their arguments, which are similar to those of strict funcons (see §8.1.4), and the term constructors have rules that apply the type constructor once their arguments have been evaluated. As an example, we give *typower*:

$$\frac{\mathit{Xs} \rightsquigarrow^* X', Y' \quad \mathit{types}, X' \overset{\sim}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \mathit{naturals}, Y' \overset{\sim}{\Rightarrow}_{ty} \mathit{true}\langle\rangle}{\mathit{typower}(\mathit{Xs}) \rightsquigarrow \mathit{typower}[X', Y']}\ \text{(MEDIUM)} \tag{8.23}$$

$$\frac{\mathit{Xs} \rightsquigarrow^* \mathit{Xs}' \quad \mathit{Xs}' \longrightarrow \mathit{Xs}''}{\mathit{typower}(\mathit{Xs}) \longrightarrow \mathit{typower}(\mathit{Xs}'')}\ \text{(MEDIUM)} \tag{8.24}$$

The rules for the small-step evaluation of term sequences are given in §8.1.3.

Remarks on ambiguous type sequences Nested applications of the type operators can result in 'ambiguous types'. For example, there are two ways to conclude that **true** is a member of the type (**booleans**$^?$, **booleans**$^?$). In Rule (8.20) it is checked whether X is of type T and whether Xs is of type $\mathit{tyseq}\langle \mathit{Ts}\rangle$. If T is formed by an application of *tyopt* then T can be ignored if X, Xs is of type $\mathit{tyseq}\langle \mathit{Ts}\rangle$. If T is formed by an application of *tystar* then any prefix of X, Xs may be of type T and the remaining suffix of type $\mathit{tyseq}\langle \mathit{Ts}\rangle$. This source of ambiguity is similar to that of a pattern sequence with several sequence variables. As given above, the rules for *tyseq*, *tystar*, *typlus*, and *typower* assume that the arguments of type operators are types of singleton sequences. As discussed in Section 8.5, this prohibits us from translating certain funcon definitions. A complete implementation of

the type operators of CBS is discussed in Section 9.3.

8.1.3 Computations

We define the relation $\dashrightarrow$, the iterative closure of $\longrightarrow$, which captures finite computations. A finite computation of n steps consists of $n+2$ configurations $\lambda_0, \lambda_1 \ldots \lambda_n, \lambda'_n$, where λ'_n is a value. At the k-th step of the computation, configuration λ_{k-1}, found by the previous steps, is rewritten to λ'_{k-1} (i.e. $\lambda_{k-1} \rightsquigarrow^* \lambda'_{k-1}$) and λ'_{k-1} transitions to λ_k (i.e. $\lambda'_{k-1} \longrightarrow \lambda_k$), the input of the next step (if any). The last step of the computation produces the configuration λ_n for which holds that $\lambda_n \rightsquigarrow^* \lambda'_n$. This description is formalised by the following rules[1]:

$$\frac{Xp \rightsquigarrow^* Vs \qquad tystar(values), Vs \stackrel{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{Xp \dashrightarrow Vs}\ \text{(LOWER)} \tag{8.25}$$

$$\frac{X \rightsquigarrow^* Xs \qquad Xs \longrightarrow Xs'}{X, Xp \longrightarrow Xs', Xp}\ \text{(MEDIUM)} \tag{8.27}$$

$$\frac{X \rightsquigarrow^* Xs \qquad Xs \longrightarrow Xs' \qquad Xs' \dashrightarrow Vs}{X \dashrightarrow Vs}\ \text{(MEDIUM)} \tag{8.26}$$

$$\frac{X \rightsquigarrow^* Vs \qquad tystar(values), Vs \stackrel{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \qquad Xp \rightsquigarrow^* Xs \qquad Xs \longrightarrow Xs'}{X, Xp \longrightarrow Vs, Xs'}\ \text{(MEDIUM)} \tag{8.28}$$

Each application of Rule (8.26) in a proof of $\lambda_0 \dashrightarrow \lambda'_n$ corresponds to a step in the computation from λ_0 to λ'_n. Rules (8.27) and (8.28) enable a single computational step within a sequence of terms[2].

8.1.4 Funcon Signatures

Funcon definitions consist of a signature and a collection of rules. Via parameter declarations, the signature of a funcon definition determines the types of the arguments that can be received by the funcon. This information can be used for statically checking funcon translations. For the operational behaviour of funcon terms, only the strictness of the parameters is relevant, determining whether arguments should be evaluated.

[1] We can have $X \dashrightarrow \#$ and $Xp \rightsquigarrow^* \#$. Because the empty sequence is not terminal, Rules (8.25) and (8.28) contain $tystar(values), Vs \stackrel{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle$ rather than **is-terminal**($\longrightarrow$, Vs).

[2] The terms are evaluated in a left-to-right fashion.

As examples, consider the signatures of **scope** and **interleave-map**:

$$\mathit{Funcon}\ \textbf{scope}(E : \textbf{environments}, X : \Rightarrow T) : \Rightarrow T \tag{8.29}$$

$$\mathit{Funcon}\ \textbf{interleave-map}(X : T \Rightarrow T', V^* : (T)^*) : \Rightarrow (T)^* \tag{8.30}$$

The parameter declarations in a signature are formed by a sequence of variable names[3], each followed by a colon and a sort. The last variable in the sequence may be a sequence variable, recognised by a super-script *, $^+$, or $^?$. The parameter list of a signature is thus divided in two parts: the initial part consisting of zero or more basic variables and a final part consisting of an optional sequence variable. A funcon with n initial parameters[4] must receive n or more arguments. The first n arguments are matched against the initial parameters, any remaining arguments are matched against the optional final parameter.

A sort is either a type, is of the form $\Rightarrow T_2$, or is of the form $T_1 \Rightarrow T_2$, with T_1 and T_2 types. A parameter is strict if its sort is a type, i.e. if it does not have a top-level[5] occurrence of $\Rightarrow$ (we ignore the meaning of the sort operator $\Rightarrow$ here). In the examples above, the first parameter of **scope** and the final parameter of **interleave-map** are strict.

The strictness of parameters determines whether congruence rules, that perform computational steps on arguments, are implicitly present in a CBS specification. In the generated IML specification, congruence rules are explicit. If there are m strict parameters, m congruence rules are generated. If f is the term constructor introduced for a particular funcon whose signature has n initial parameters and a final parameter, then the congruence rule generated for the k-th argument, with $k \leqslant n$, is of the form:

$$\frac{X_k \rightsquigarrow^* X'_k \quad X'_k \longrightarrow X''_k}{f(X_1, \ldots, X_k, X_{k+1}, \ldots, X_n, Xs) \longrightarrow f(X_1, \ldots, X''_k, X_{k+1}, \ldots, X_n, Xs)}\ \texttt{(MEDIUM)}$$

If the signature of f does not have a final parameter, then Xs does not appear in the source and target of the conclusion. If the signature of f does have a final parameter, and if it is

[3] We ignore wildcards in this description, which can replaced by fresh variables.

[4] We use 'parameters' as a shorthand for 'variables declared in a signature', arguments are terms given as part of an application, and a basic variable is a placeholder for exactly one term.

[5] Sorts can be used to construct types as well, e.g. sorts appear as the argument of **abstractions**.

strict, then a congruence rule of the following form is generated:

$$\frac{Xs \rightsquigarrow^* Xs' \quad Xs' \longrightarrow Xs''}{f(X_1, \ldots, X_n, Xs) \longrightarrow f(X_1, \ldots, X_n, Xs'')} \text{ (MEDIUM)}$$

As describe above, more than one congruence rule may be applicable at a time. If desired, it is possible to specify a left-to-right evaluation order for the arguments by including a side-condition $values, X_j \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle$, for all $1 \leqslant j < k$, with X_j a strict parameter.

The congruence rules for **scope** and **interleave-map** are as follows:

$$\frac{X_1 \rightsquigarrow^* X_1' \quad X_1' \longrightarrow X_1''}{scope(X_1, X_2) \longrightarrow scope(X_1'', X_2)} \text{ (MEDIUM)} \tag{8.31}$$

$$\frac{Xs \rightsquigarrow^* Xs' \quad Xs' \longrightarrow Xs''}{interleave\text{-}map(X_1, Xs) \longrightarrow interleave\text{-}map(X_1, Xs'')} \text{ (MEDIUM)} \tag{8.32}$$

8.1.5 Built-in Operations and Built-in Types

Some CBS definitions are tagged with the keyword *Built-in*, indicating that these definitions can be ignored by the compiler and are implemented manually. The most common built-in definitions are for types and operations on values of these types. As mentioned in Section 7.4, CBS' built-in types and operations are implemented in `funcons-values`, a Haskell package underlying the IML interpreter. In this subsection we discuss the rules required to access these built-in types and operations.

For each operation in `funcons-values` there is a term constructor whose semantics are to evaluate its arguments and apply the operation. For example, consider the following CBS definition and IML rules for the value operation **integer-add** (note that the CBS definition has no rules):

Built-in Type **integers** (8.33)

Built-in Funcon **integer-add**(_ : **integers***) : ⇒**integers** (8.34)

$$\frac{Xs \rightsquigarrow^* Xs' \quad tystar(values), Xs' \Rightarrow_{ty} true\langle\rangle}{integer\text{-}add(Xs) \rightsquigarrow \texttt{integer-add}(Xs')}\ (10) \tag{8.35}$$

$$\frac{Xs \rightsquigarrow^* Xs' \quad Xs' \longrightarrow Xs''}{integer\text{-}add(Xs) \longrightarrow integer\text{-}add(Xs'')}\ (10) \tag{8.36}$$

All value operations are strict in all their arguments and have a congruence rule like (8.36). If Xs is a sequence of length greater than one, establishing the premise of Rule (8.36) involves applying either one of the Rules (8.27) or (8.28). All strict funcons have a similar congruence rule (for example **variant**, given later).

Funcon **integer-add** is variadic; it receives an integer sequence as an argument. The sequence may be the result of rewriting a single term into a sequence, or rewriting several terms into a concatenation of sequences. For example, the following terms all compute 7:

integer-add(1, 2, 3)

integer-add(**tuple-elements**(**tuple**(1, 2, 3)))

integer-add(**tuple-elements**(**tuple**(1, 2)), **tuple-elements**(**tuple**(1)))

The arguments of an operation with a fixed number of arguments, such as the binary **integer-subtract**, may also result from rewriting. For example, **integer-subtract**(**tuple-elements**(**tuple**(2,1))) evaluates to 1. The IML rules generated for **integer-subtract** are like those for **integer-add**:

$$\frac{Xs \rightsquigarrow^* Xs' \quad tystar(values), Xs' \Rightarrow_{ty} true\langle\rangle}{integer\text{-}subtract(Xs) \rightsquigarrow \texttt{integer-subtract}(Xs')}\ (10) \tag{8.37}$$

$$\frac{Xs \rightsquigarrow^* Xs' \quad Xs' \longrightarrow Xs''}{integer\text{-}subtract(Xs) \longrightarrow integer\text{-}subtract(Xs'')}\ (10) \tag{8.38}$$

The operation `integer-subtract` is only defined on two (integer) arguments. Therefore Rule (8.37) is not applicable if Xs' is not a sequence of length 2.

In general, if all the parameters of a funcon are strict, then its arguments need to be

rewritten before pattern matching occurs in any of the user-defined rules. Sections 8.3 and 8.4 discuss the translation of user-defined CBS rules into IML rules. The next section shows how the $\Rightarrow_{ty}$-relation is extended by generating IML rules from CBS type definitions.

8.2 Types

CBS type definitions come in two forms: type synonyms and algebraic datatypes.

8.2.1 Type Synonyms

A type synonym introduces a shorthand for a type. Consider, for example, the following definition of the type **environments**:

$$\mathit{Type}\ \textbf{environments} \rightsquigarrow \textbf{maps}(\textbf{identifiers}, \textbf{values}^{?}) \tag{8.39}$$

The type definition above is syntactic sugar for the following funcon definition:

$$\mathit{Funcon}\ \textbf{environments} : \Rightarrow\textbf{types} \tag{8.40}$$

$$\mathit{Rule}\ \textbf{environments} \rightsquigarrow \textbf{maps}(\textbf{identifiers}, \textbf{values}^{?}) \tag{8.41}$$

Because type synonym definitions are syntactic sugar, the CBS compiler does not generate IML code for type synonyms directly. A type defined as a synonym may also have parameters, which are translated in a straightforward manner.

8.2.2 Algebraic Data Types

A datatype definition introduces a new type of which each value is the result of applying one of its constructors. For example, the type **booleans** is defined as follows:

$$\mathit{Datatype}\ \textbf{booleans}\ ::=\ \textbf{true} \mid \textbf{false} \tag{8.42}$$

For **booleans**, the following IML rules are generated:

$$\frac{}{booleans() \rightsquigarrow booleans[]}\ \text{(MEDIUM)} \tag{8.43}$$

$$\frac{}{true() \rightsquigarrow true\langle\rangle}\ \text{(MEDIUM)} \tag{8.44}$$

$$\frac{}{false() \rightsquigarrow false\langle\rangle}\ \text{(MEDIUM)} \tag{8.45}$$

$$\frac{}{booleans[], true\langle\rangle \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.46}$$

$$\frac{}{booleans[], false\langle\rangle \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.47}$$

Rules (8.43), (8.44), and (8.45) rewrite applications of the (nullary) term constructors *booleans*, *true*, and *false* to applications of their corresponding type or value constructors.

Datatype definitions can directly include another type as a sub-type. For example, **strings** are included in the type **identifiers**. An **identifier** may also be a simpler **identifier** tagged with an arbitrary value (e.g. with the **identifier** of a namespace):

$$\textit{Datatype}\ \textbf{identifiers}\ ::=\ \{\_ : \textbf{strings}\} \mid \textbf{identifier-tagged}(\_ : \textbf{identifiers}, \_ : \textbf{values}) \tag{8.48}$$

The CBS compiler generates the following IML rules for type-checking **identifiers**:

$$\frac{strings, X \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{identifiers[], X \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.49}$$

$$\frac{identifiers, X_1 \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \quad values, X_2 \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{identifiers[], identifier\text{-}tagged\langle X_1, X_2\rangle \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.50}$$

An algebraic datatype may have type parameters, and its constructors may have arguments restricted to certain types, possibly depending on the type parameters. For example, the definition of **variants** specifies that the second argument of its constructor **variant** is a value of a given type T:

$$\textit{Datatype}\ \textbf{variants}(T)\ ::=\ \textbf{variant}(\_ : \textbf{identifiers}, \_ : T) \tag{8.51}$$

The rule for type-checking **variants** involves premises for type-checking arguments:

$$\frac{identifiers, X_1 \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \quad T, X_2 \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{variants[T], variant\langle X_1, X_2\rangle \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.52}$$

Funcons **variants** and **variant** are both strict; the term constructors *variants* and *variant* have a congruence rule to evaluate their argument(s). Once the arguments are evaluated, the corresponding type and value constructor is applied:

$$\frac{Xs \rightsquigarrow^* Xs' \qquad Xs' \longrightarrow Xs''}{variants(Xs) \longrightarrow variants(Xs'')}\ \text{(MEDIUM)} \tag{8.53}$$

$$\frac{Xs \rightsquigarrow^* Xs' \qquad Xs' \longrightarrow Xs''}{variant(Xs) \longrightarrow variant(Xs'')}\ \text{(MEDIUM)} \tag{8.55}$$

$$\frac{Xs \rightsquigarrow^* Vs}{tystar(values), Vs \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{variants(Xs) \longrightarrow variants[Vs]}\ \text{(MEDIUM)} \tag{8.54}$$

$$\frac{Xs \rightsquigarrow^* Vs}{tystar(values), Vs \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{variant(Xs) \longrightarrow variant\langle Vs\rangle}\ \text{(MEDIUM)} \tag{8.56}$$

8.2.3 Value-dependent Types – Example

The parameters of types are not restricted to types. For example, the type **bit-vectors** receives a natural number as argument, determining the size of the vector:

Type **bits** $\rightsquigarrow$ **booleans** (8.57)

Type **bytes** $\rightsquigarrow$ **bit-vectors**(8) (8.58)

Datatype **bit-vectors**(N : **natural-numbers**) ::= **bit-vector**(_ : **bits**$^{\wedge N}$) (8.59)

The following rule is generated for type-checking **bit-vectors**:

$$\frac{typower(bits, N), Xs \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{bit\text{-}vectors[N], bit\text{-}vector\langle Xs\rangle \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.60}$$

8.2.4 Composite Built-in Types

Built-in funcons **maps** and **sets** receive types as arguments. Since the arguments may be user-defined types, Rule (8.6) is not adequate for type-checking sets and maps. For this reason, we introduce type constructors *maps* and *sets* with specialised rules for type-checking. The following rule is for type-checking sets, applying `set-elements` and *tystar*:

$$\frac{\texttt{set-elements}(S) \rhd Vs \quad tystar[T], Vs \Rightarrow_{ty} true\langle\rangle}{sets[T], S \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.61}$$

The following rule is for type-checking maps and relies on `map-elements` returning a sequence of associations as tuples:

$$\frac{\texttt{map-elements}(M) \rhd Vs \qquad tystar[tuples[K,V]], Vs \Rightarrow_{ty} true\langle\rangle}{maps[K,V], M \Rightarrow_{ty} true\langle\rangle}\ \text{(MEDIUM)} \tag{8.62}$$

8.3 Rewrite Rules

The conclusion of a rewrite rule translates directly into IML: CBS terms are translated to IML terms, CBS patterns to IML patterns, CBS variables to IML variables, etc. This section focusses on the translation of the conditions that can appear in rewrite rules.

8.3.1 Type-checking

A condition $X : T$ rewrites X to a value X', T to a type T', and checks whether X' is of type T'. Thus $X : T$ translates to the premise $[\![T]\!], [\![X]\!] \stackrel{\sim}{\Rightarrow}_{ty} true\langle\rangle$, where $[\![X]\!]$ and $[\![T]\!]$ are the translations of X and T. Consider the definition of **is-in-type**:

$$Funcon\ \textbf{is-in-type}(V : \textbf{values}, T : \textbf{types}) \tag{8.63}$$

$$Rule\ \frac{V : T}{\textbf{is-in-type}(V : \textbf{values}, T : \textbf{values}) \rightsquigarrow \textbf{true}} \tag{8.64}$$

$$Rule\ \frac{V : {\sim}T}{\textbf{is-in-type}(V : \textbf{values}, T : \textbf{values}) \rightsquigarrow \textbf{false}} \tag{8.65}$$

Besides congruence rules, the following IML rules are generated from the above definition:

$$\frac{Xs \rightsquigarrow^* V, T \quad types, T \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \quad values, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \qquad T, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{is\text{-}in\text{-}type(Xs) \rightsquigarrow true\langle\rangle}\ \texttt{(MEDIUM)} \tag{8.66}$$

$$\frac{\begin{array}{c} Xs \rightsquigarrow^* V, T \quad types, T \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \quad values, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \\ tyneg(T), V \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \end{array}}{is\text{-}in\text{-}type(Xs) \rightsquigarrow false\langle\rangle}\ \texttt{(MEDIUM)} \tag{8.67}$$

The argument sequence Xs is rewritten to a sequence of length two of which the first component must be a value and the second component a type. The premise $types, T \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle$ is a side-effect of translating the CBS pattern T : **types** of which the annotation : **types** determines that T must be a value of type **types**. Similarly, the premise $values, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle$ is a side-effect of translating the pattern V : **values**.

8.3.2 Equality

A condition $X_1 == X_2$ rewrites both operands to values and checks whether they are structurally equal. The operation `is-equal`, built-in to IML, checks whether two IML terms are structurally equal. A condition $X_1 == X_2$ is translated into $\texttt{is-equal}(X_1', X_2') \triangleright true\langle\rangle$ with X_1' the result of rewriting $[\![X_1]\!]$ and X_2' of rewriting $[\![X_2]\!]$. Extra conditions are generated to ensure that X_1' and X_2' are values. As an example, we give **is-equal**. The binary funcon **is-equal** checks whether its arguments are **ground-values**, i.e. **values** that do not contain **abstractions**:

$$Funcon\ \textbf{is-equal}(V : \textbf{values}, W : \textbf{values}) \tag{8.68}$$

$$Rule\ \frac{V == W}{\textbf{is-equal}(V : \textbf{ground-values}, W : \textbf{ground-values}) \rightsquigarrow \textbf{true}} \tag{8.69}$$

$$Rule\ \frac{V =/= W}{\textbf{is-equal}(V : \textbf{ground-values}, W : \textbf{ground-values}) \rightsquigarrow \textbf{false}} \tag{8.70}$$

$$Rule\ \textbf{is-equal}(V : \sim\textbf{ground-values}, W : \textbf{ground-values}) \rightsquigarrow \textbf{false} \tag{8.71}$$

$$Rule\ \textbf{is-equal}(V : \textbf{ground-values}, W : \sim\textbf{ground-values}) \rightsquigarrow \textbf{false} \tag{8.72}$$

The following IML rules are generated from the rewrite rules above:

$$\frac{Xs \rightsquigarrow^* V, W \qquad \mathit{ground\text{-}values}, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \mathit{ground\text{-}values}, W \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \texttt{is-equal}(V, W) \rhd \mathit{true}\langle\rangle}{\mathit{is\text{-}equal}(Xs) \rightsquigarrow \mathit{true}\langle\rangle}\ \text{(MEDIUM)} \tag{8.73}$$

$$\frac{Xs \rightsquigarrow^* V, W \qquad \mathit{ground\text{-}values}, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \mathit{ground\text{-}values}, W \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \texttt{is-equal}(V, W) \rhd \mathit{false}\langle\rangle}{\mathit{is\text{-}equal}(Xs) \rightsquigarrow \mathit{false}\langle\rangle}\ \text{(MEDIUM)} \tag{8.74}$$

$$\frac{Xs \rightsquigarrow^* V, W \qquad \mathit{tyneg}(\mathit{ground\text{-}values}), V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \mathit{ground\text{-}values}, W \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle}{\mathit{is\text{-}equal}(Xs) \rightsquigarrow \mathit{false}\langle\rangle}\ \text{(MEDIUM)} \tag{8.75}$$

$$\frac{Xs \rightsquigarrow^* V, W \qquad \mathit{ground\text{-}values}, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \qquad \mathit{tyneg}(\mathit{ground\text{-}values}), W \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle}{\mathit{is\text{-}equal}(Xs) \rightsquigarrow \mathit{false}\langle\rangle}\ \text{(MEDIUM)} \tag{8.76}$$

The condition $X_1 =/= X_2$ is translated into an application of `is-equal` with result pattern $\mathit{false}\langle\rangle$. Note that in this example there is no need to generate conditions $\mathit{values}, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle$ and $\mathit{values}, W \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle$, as suggested earlier, because of the conditions generated for the pattern annotations.

8.3.3 Rewrite Conditions

Rewrite rules have $\rightsquigarrow$-premises, which perform as many rewrites as possible to a term, but not necessarily to a value. A $\rightsquigarrow$-premise in CBS thus translates into a $\rightsquigarrow^*$-premise in IML. Consider the following definition of **if-true-else**, and the IML rules generated for it:

$$\mathit{Funcon}\ \textbf{if-true-else}(\_ : \textbf{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T \tag{8.77}$$

$$\mathit{Rule}\ \frac{B \rightsquigarrow \textbf{true}}{\textbf{if-true-else}(B, X, \_) \rightsquigarrow X} \tag{8.78}$$

$$\mathit{Rule}\ \frac{B \rightsquigarrow \textbf{false}}{\textbf{if-true-else}(B, \_, Y) \rightsquigarrow Y} \tag{8.79}$$

$$\frac{B \rightsquigarrow^* \mathit{true}\langle\rangle}{\mathit{if\text{-}true\text{-}else}(B, X, Y) \rightsquigarrow X}\ \text{(MEDIUM)} \quad (8.80) \qquad \frac{B \rightsquigarrow^* \mathit{false}\langle\rangle}{\mathit{if\text{-}true\text{-}else}(B, X, Y) \rightsquigarrow Y}\ \text{(MEDIUM)} \quad (8.81)$$

As a pattern on the right-hand side of $B \rightsquigarrow$ **true**, **true** is translated to an IML pattern in which *true* occurs as a value constructor rather than a term constructor (and similarly for **false**). In general, when translating CBS patterns to IML patterns, occurrences of constructors are translated into their IML value constructor equivalent. However, before testing whether a term matches a pattern, the term needs to be fully rewritten. To achieve this, CBS patterns with occurrences of value constructors translate into variables, with conditions on those variables generated as a side-effect (similar to type annotations in §8.3.1). Consider the following alternative rules defining **if-true-else**:

$$\mathit{Rule}\ \textbf{if-true-else}(\textbf{true}, X, \_) \rightsquigarrow X \quad (8.82)$$

$$\mathit{Rule}\ \textbf{if-true-else}(\textbf{false}, \_, Y) \rightsquigarrow Y \quad (8.83)$$

Translating patterns in the way described above, the same IML rules are generated for both alternative sets of rules for **if-true-else**.

8.4 Entities

Computation rules differ from rewrite rules in that their conclusion is a $\longrightarrow$-transition, that $\longrightarrow$-transitions can occur as premises, and that $\longrightarrow$-transitions may refer to entities.

CBS inference rules are instances of I-MSOS rules (see §6.3.3) and involve several *classes* of entities. Each class has a category associated with it, together with a concrete notation in rules — the 'reference style' — for identifying morphisms of the category. Several entities can belong to the same class and entities are declared in CBS specifications to indicate to which class they belong. Each entity class can be explained by giving the details of the underlying category and by explaining the connection between the reference style and morphisms of the category. Instead, we introduce classes informally and explain their operational semantics via the translation to IML. This section describes how $\longrightarrow$-transitions are translated to IML, focussing specifically on the entity references of CBS' entity classes. We discuss mutable,

contextual, output, and control entities.

From this perspective, IML has only one entity class, which corresponds to the read-write entities of MSOS [Mosses, 2004], called mutable entities in CBS. The process of translating references to contextual, output, and control entities thus explains how the entities of these classes can be seen as mutable entities. The Haskell Funcon Framework of Chapter 9 supports interactive input and 'bidirectional' control entities, which this chapter ignores.

8.4.1 Mutable Entities

A mutable entity is declared as follows, taking the entity store as an example:

$$\mathit{Entity}\ \langle \_ \,,\, \mathsf{store}(\_ : \mathbf{stores}) \rangle \longrightarrow \langle \_ \,,\, \mathsf{store}(\_ : \mathbf{stores}) \rangle \tag{8.84}$$

$$\mathit{Type}\ \mathbf{stores} \rightsquigarrow \mathbf{maps}(\mathbf{locations}, \mathbf{values}^?) \tag{8.85}$$

The declaration reveals the reference style of mutable entities: whenever a transition refers to mutable entities, the source and target of the transition are a program fragment (_ in the declaration) followed by the names of referred entities with their values inside parenthesis (and both source and target are surrounded by ⟨ and ⟩).

Translating the entity references of mutable entities simply involves translating CBS terms to IML terms and CBS patterns to IML patterns. For each mutable entity, the left-hand side of a conclusion contains a pattern and a the right-hand side a term. In a premise this is reversed. However, all the funcons in the beta-release of CBS that interact with mutable entities are defined by rules without premises.

The following example shows the translation for the rules of **assign**:

$$\mathit{Funcon}\ \mathbf{assign}(\_ : \mathbf{variables}, \_ : \mathbf{values}) : \Rightarrow \mathbf{null\text{-}type} \tag{8.86}$$

$$\mathit{Rule}\ \frac{\mathbf{and}(\mathbf{is\text{-}in\text{-}set}(L, \sigma), \mathbf{is\text{-}in\text{-}type}(V, T)) \rightsquigarrow \mathbf{true} \qquad \mathbf{map\text{-}override}(\{L \mapsto V\}, \sigma) \rightsquigarrow \sigma'}{\langle \mathbf{assign}(\mathbf{variable}(L, T), V : \mathbf{values}) \,,\, \mathsf{store}(\sigma) \rangle \longrightarrow \langle \mathbf{null} \,,\, \mathsf{store}(\sigma') \rangle} \tag{8.87}$$

$$\mathit{Rule}\ \frac{\mathbf{and}(\mathbf{is\text{-}in\text{-}set}(L, \sigma), \mathbf{is\text{-}in\text{-}type}(V, T)) \rightsquigarrow \mathbf{false}}{\langle \mathbf{assign}(\mathbf{variable}(L, T), V : \mathbf{values}) \,,\, \mathsf{store}(\sigma) \rangle \longrightarrow \langle \mathbf{fail} \,,\, \mathsf{store}(\sigma) \rangle} \tag{8.88}$$

$$\frac{X \rightsquigarrow^* \mathit{variable}\langle L, T\rangle \qquad \mathit{and}(\mathit{is\text{-}in\text{-}set}(L, \mathit{dom}(\mathit{Sig})), \mathit{is\text{-}in\text{-}type}(V, T)) \rightsquigarrow^* \mathit{true}\langle\rangle \qquad \mathit{map\text{-}override}(\mathit{map}(\mathit{tuple}(L, V)), \mathit{Sig}) \rightsquigarrow^* Y \qquad \mathit{values}, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle}{\mathit{assign}(X, V), \texttt{store} = \mathit{Sig} \longrightarrow \mathit{null}, \texttt{store} = Y}\ \text{(MEDIUM)} \tag{8.89}$$

$$\frac{\mathit{values}, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \quad X \rightsquigarrow^* \mathit{variable}\langle L, T\rangle \quad \mathit{and}(\mathit{is\text{-}in\text{-}set}(L, \mathit{dom}(\mathit{Sig})), \mathit{is\text{-}in\text{-}type}(V, T)) \rightsquigarrow^* \mathit{false}\langle\rangle}{\mathit{assign}(X, V), \texttt{store} = \mathit{Sig} \longrightarrow \mathit{fail}, \texttt{store} = \mathit{Sig}}\ \text{(MEDIUM)} \tag{8.90}$$

8.4.2 Contextual Entities

A contextual entity, or read-only entity in MSOS, is declared as follows:

$$\mathit{Entity}\ \mathsf{environment}(\_ : \textbf{environments}) \vdash \_ \longrightarrow \_ \tag{8.91}$$

The value held by a contextual entity does not change as part of a transition. A reference to a contextual entity therefore consists of only one value and appears before a turnstile. In the translation to IML, we translate the reference for inclusion in both the left-hand side and right-hand side of the IML transition, ensuring the entity does not change by the transition. There is no need to generate an entity reference for the right-hand side of premises. As an example, consider the definition of **scope** and its translation:

$$\mathit{Funcon}\ \textbf{scope}(\_ : \textbf{environments}, \_ : \Rightarrow T) : \Rightarrow T \tag{8.92}$$

$$\mathit{Rule}\ \frac{\textbf{map-override}(\rho_1, \rho_0) \rightsquigarrow \rho_2 \qquad \mathsf{environment}(\rho_2) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \textbf{scope}(\rho_1 : \textbf{environments}, X) \longrightarrow \textbf{scope}(\rho_1, X')} \tag{8.93}$$

$$\mathit{Rule}\ \textbf{scope}(\_ : \textbf{environments}, V : T) \rightsquigarrow V \tag{8.94}$$

$$\frac{\begin{array}{c} environments, Rho_1 \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \quad map\text{-}override(Rho_1, Rho_2) \rightsquigarrow^* Y \\ X \rightsquigarrow^* Z \quad Z, \texttt{environment} = Y \longrightarrow X' \end{array}}{scope(Rho_1, X), \texttt{environment} = Rho_0 \longrightarrow scope(Rho_1, X'), \texttt{environment} = Rho_0} \text{(MEDIUM)} \tag{8.95}$$

$$\frac{environments, X \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle \qquad values, V \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{scope(X, V) \rightsquigarrow V} \text{(MEDIUM)} \tag{8.96}$$

In the IML rule generated for (8.94), the wildcard is replaced by a fresh variable in order to check the type annotation as a condition. The unknown type T translates to *values*.

8.4.3 Output Entities

Currently, the only output entity is standard-out, which models program output as a sequence of values. It is declared as follows:

$$Entity \_ \xrightarrow{\text{standard-out!}(\_:\textbf{values}^*)} \_ \tag{8.97}$$

Output entities hold sequences of values. A reference to an output entity consists of a single value sequence which appears on the arrow behind an exclamation mark. (Although not shown here, a question mark is used for input entities, the dual of output entities.) Funcon **print** is defined as follows:

$$Funcon\ \textbf{print}(\_ : \textbf{values}^*) : \Rightarrow \textbf{null-type} \tag{8.98}$$

$$Rule\ \textbf{print}(V^* : \textbf{values}^*) \xrightarrow{\text{standard-out!}(V^*)} \textbf{null} \tag{8.99}$$

In the generated IML, the sequence $[\![V^*]\!]$ is appended to the output already held by the output entity. The following IML rule is generated from the CBS rule given above:

$$\frac{tystar(values), Vs \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{print(Vs), \texttt{standard-out} = Y \longrightarrow null, \texttt{standard-out} = Y, Vs} \text{(MEDIUM)} \tag{8.100}$$

There are currently no examples of funcons with one or more rules that contain a premise referring to an output entity.

8.4.4 Control Entities

Control entities hold a single value or no value at all. They are used to model control flow operators in a general sense. For example, exception handling, return statements, pattern match failure, and delimited continuations [Sculthorpe et al., 2016] can all be specified with control entities. The presence of a value in a control entity signals abrupt termination in the evaluation of some sub-term of a larger term. The value is referred to as a signal. The larger term may respond to the signal and continue evaluating another subterm, for example an exception-handler. The abrupted control entity is defined as follows:

$$\mathit{Entity}\ \_ \xrightarrow{\mathsf{abrupted}(\_:\mathbf{values}^?)} \_ \tag{8.101}$$

The reference style is similar to output entities, but without the exclamation mark. If a reference to a control entity mentions no value, this indicates the continued absence of a signal. If it does, this indicates the appearance of a signal. As examples of 'emitting' and 'receiving' signals, we use **abrupt** and **handle-abrupt**, interacting with abrupted:

$$\mathit{Funcon}\ \mathbf{abrupt}(\_ : \mathbf{values}) : \Rightarrow \mathbf{empty\text{-}type} \tag{8.102}$$

$$\mathit{Rule}\ \mathbf{abrupt}(V : \mathbf{values}) \xrightarrow{\mathsf{abrupted}(V)} \mathbf{stuck} \tag{8.103}$$

$$\mathit{Funcon}\ \mathbf{handle\text{-}abrupt}(\_ : S \Rightarrow T, \_ : X \Rightarrow T) : S \Rightarrow T \tag{8.104}$$

$$\mathit{Rule}\ \frac{X \xrightarrow{\mathsf{abrupted}(V)} \_}{\mathbf{handle\text{-}abrupt}(X, H) \xrightarrow{\mathsf{abrupted}()} \mathbf{give}(V, H)} \tag{8.105}$$

$$\mathit{Rule}\ \frac{X \xrightarrow{\mathsf{abrupted}()} X'}{\mathbf{handle\text{-}abrupt}(X, H) \xrightarrow{\mathsf{abrupted}()} \mathbf{handle\text{-}abrupt}(X', H)} \tag{8.106}$$

$$\mathit{Rule}\ \mathbf{handle\text{-}abrupt}(V : T, \_) \rightsquigarrow V \tag{8.107}$$

The following IML rules are generated for the CBS rules given above:

$$\frac{V \rightsquigarrow^* V' \qquad values, V' \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{abrupt(V), \texttt{abrupted} = \# \longrightarrow stuck, \texttt{abrupted} = V'} \text{(MEDIUM)} \tag{8.108}$$

$$\frac{X \rightsquigarrow^* X_1 \qquad X_1, \texttt{abrupted} = \# \longrightarrow X_2, \texttt{abrupted} = V}{handle\text{-}abrupt(X, H), \texttt{abrupted} = \# \longrightarrow give(V, H), \texttt{abrupted} = \#} \text{(MEDIUM)} \tag{8.109}$$

$$\frac{X \rightsquigarrow^* X_1 \qquad X_1, \texttt{abrupted} = \# \longrightarrow X_2, \texttt{abrupted} = \#}{handle\text{-}abrupt(X, H), \texttt{abrupted} = \# \longrightarrow handle\text{-}abrupt(X_2, H), \texttt{abrupted} = \#} \text{(MEDIUM)} \tag{8.110}$$

$$\frac{V \rightsquigarrow^* V' \qquad values, V' \overset{\rightsquigarrow}{\Rightarrow}_{ty} true\langle\rangle}{handle\text{-}abrupt(V, H) \rightsquigarrow V'} \text{(MEDIUM)} \tag{8.111}$$

It follows from the way transitions with control entity references are generated that there are no transitions in which a signal disappears (e.g. the source has $\texttt{abrupted} = V$ and the target has $\texttt{abrupted} = \#$) or remains (e.g. source and target have $\texttt{abrupted} = V$). A configuration in which a value for a control entity is held is thus stuck. To accept such a configuration as the final configuration in a computation, we declare that all values cause termination in control entities. For example, in the case of abrupted, we add the declaration:

$$\textbf{terminal}(\texttt{abrupted}, \textbf{values}) \tag{8.112}$$

The following rule is necessary to ensure that $\dashrightarrow$ still captures all finite computations in the transition system for $\longrightarrow$:

$$\frac{}{Xs, \texttt{abrupted} = V \dashrightarrow Xs, \texttt{abrupted} = V} \text{(MEDIUM)} \tag{8.113}$$

8.5 Overcoming Translation Restrictions

Left-to-right Evaluation The funcons **left-to-right** and **right-to-left** cannot be translated to IML because they involve a rule in which two sequence variables appear together in

a pattern sequence, causing ambiguity (see §7.3.2). However, computational steps over term sequences transition IML terms in a left-to-right fashion (see §8.1.3). It is therefore easy to give built-in IML rules for **left-to-right**. Funcon **left-to-right** is defined by the following CBS rules:

$$\mathit{Rule}\ \frac{Y \longrightarrow Y'}{\textbf{left-to-right}(V^* : (T^*), Y, Z^*) \longrightarrow \textbf{left-to-right}(V^*, Y', Z^*)} \tag{8.114}$$

$$\mathit{Rule}\ \textbf{left-to-right}(V^* : (T^*)) \rightsquigarrow V^* \tag{8.115}$$

The following IML rules capture the behaviour of **left-to-right**:

$$\frac{Xs \rightsquigarrow^* Xs' \qquad Xs' \longrightarrow Xs''}{\mathit{left\text{-}to\text{-}right}(Xs) \longrightarrow \mathit{left\text{-}to\text{-}right}(Xs'')}\ \texttt{(MEDIUM)} \tag{8.116}$$

$$\frac{Vs \rightsquigarrow^* Vs' \qquad \mathit{tystar}(\mathit{values}),\ Vs' \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle}{\mathit{left\text{-}to\text{-}right}(Vs) \rightsquigarrow Vs'}\ \texttt{(MEDIUM)} \tag{8.117}$$

By enabling **left-to-right** as a built-in, we can translate **left-to-right-map**, **left-to-right-repeat**, and **left-to-right-filter**, which are defined in terms of **left-to-right**.

Type-checking Environments In §8.2.4 we gave rules for the built-in composite types maps and sets. The rule for maps involves the type $\mathit{tuples}[K, V]$, where K and V are type arguments determining the types of the keys and values of the map. The typing rule for tuples involves the type operator *tyseq*:

$$\frac{\mathit{tyseq}(Ts),\ Vs \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle}{\mathit{tuples}[Ts], \mathit{tuple}\langle Vs\rangle \Rightarrow_{ty} \mathit{true}\langle\rangle}\ \texttt{(MEDIUM)} \tag{8.118}$$

In §8.1.2 we discussed that dynamic type-checking is correct if type constructors such as *tyseq* are only applied to types of singleton sequences. However, **maps** may be applied to a type of optional values, i.e. types containing the empty sequence. For example, in the definition of **environments** (Rule (8.41)), **maps** is applied to **identifiers** and **values?**. Type-checking environments thus involves the type $\mathit{tyseq}(\mathit{identifiers}, \mathit{tyopt}(\mathit{values}))$. The

rules for *tyseq*, however, are such that only sequences of length n can be of a type of the form $tyseq(t_1, \ldots, t_n)$. Thus, incorrectly $tyseq(identifiers, tyopt(values))$, `"x"` $\overset{\sim}{\Rightarrow}_{ty}$ $false\langle\rangle$. To remedy this specific situation, we add the following IML rule:

$$\frac{T \Rightarrow_{ty} true\langle\rangle}{tyseq[T] \Rightarrow_{ty} true\langle\rangle}\ \texttt{(MEDIUM)} \tag{8.119}$$

This rule specifies that if the empty sequence is of type T, then it is of type $tyseq[T]$.

8.6 Homogeneous Generative Meta-Programming

A language with constructs for Homogeneous Generative Meta-Programming (HGMP) enables writing code that generates code. As data, the generated code can be propagated and manipulated freely, before being inserted and evaluated in the overarching program. For example, Template Haskell [Sheard and Peyton Jones, 2002] supports HGMP at compiletime, MetaML [Taha and Sheard, 2000] at runtime, while Converge [Tratt, 2005] supports both. An overview of the features of several HGMP languages is found in [Berger et al., 2017]. In [Van Binsbergen, 2018], we defined funcons for HGMP based on earlier formalisations of HGMP by [Berger and Tratt, 2010, Berger et al., 2017]. In this section we implement the funcons introduced in [Van Binsbergen, 2018] by giving CBS and IML rules that fit within the framework of this chapter.

In [Berger et al., 2017], Berger, Tratt and Urban present a calculus for reasoning about several aspects of HGMP. Their calculus is the result of applying a semi-mechanical 'HGMPification recipe' to a standard untyped λ-calculus. The funcons of [Van Binsbergen, 2018] are directly derived from the HGMP recipe. The recipe extends languages with abstract syntax trees (ASTs) to serve as meta-representations of program fragments, and runtime and compiletime HGMP constructs.

8.6.1 Abstract Syntax Trees

There are two types of AST nodes. Firstly, an AST node can be labelled with a value v and a type τ, in which case the node has no children. Secondly, an AST node can be labelled

with a funcon f and have zero or more children. Funcons — i.e. funcon constructors — are not funcon terms, only applications of funcons are. In [Van Binsbergen, 2018], *tags* are introduced to represent funcons. Here we implement tags as strings:

$$\mathit{Type}\ \mathsf{tags} \rightsquigarrow \mathsf{strings} \tag{8.120}$$

ASTs are formalised by the following CBS datatype definition:

$$\mathit{Datatype}\ \mathsf{asts}\ ::=\ \mathsf{ast\text{-}value}(\_ : \mathsf{types}, \_ : \mathsf{values}) \mid \mathsf{ast\text{-}term}(\_ : \mathsf{tags}, \_ : \mathsf{asts}^*) \tag{8.121}$$

ASTs may be partially evaluated in the sense that where we expect an AST-node, we may find a computation instead, yielding an AST-node when evaluated. For example, the funcon term $\mathsf{give}(\mathsf{true}, \mathsf{ast\text{-}value}(\mathsf{booleans}, \mathsf{given}))$ requires evaluation to yield the AST $\mathsf{ast\text{-}value}(\mathsf{booleans}, \mathsf{true})$.

The funcons for HGMP involve three additional relations: the $\Rightarrow$-relation models a compilation phase, the $\Uparrow$-relation models the conversion from terms into their meta-representation, and the $\Downarrow$-relation models the conversion from ASTs to the terms they represent. The IML definitions of these relations that follow, apply operations built-in to the IML interpreter for constructing and deconstructing terms. The operation `term-construct` is given a string s and a sequence of arguments $x_1, \ldots, x_n$ to build the term $f(x_1, \ldots, x_n)$, where f is the constructor represented by the string s. Conversely, `term-constructor` is given a term and returns a string representing f if the given term is of the form $f(x_1, \ldots, x_n)$ (and fails otherwise, e.g. when applied to **values**). The operation `term-arguments` is given a term and returns the term sequence $x_1, \ldots, x_n$ if the given term is of the form $f(x_1, \ldots, x_n)$ (and fails otherwise). In the context of this chapter, these operations make it easy to define rules that are applicable to more than just one funcon.

The following rules for $\Downarrow$ convert ASTs to the terms they represent:

$$\frac{Vs' \Downarrow Vs \quad \texttt{term-construct}(C, Vs) \rhd X}{\mathit{ast\text{-}term}\langle C, Vs' \rangle \Downarrow X}\ \text{(MEDIUM)} \tag{8.122}$$

$$\frac{\texttt{term-construct}(C) \rhd X}{\mathit{ast\text{-}term}\langle C \rangle \Downarrow X}\ \text{(MEDIUM)} \tag{8.123}$$

give(**code**(**bound**("x")), **scope**(**bind**("x", 7), **eval**(**given**)))
→ **give**(**ast-term**("bound", **ast-value**(**identifiers**, "x")), **scope**(**bind**("x", 7), **eval**(**given**)))
→ **give**(**ast-term**("bound", **ast-value**(**identifiers**, "x")), **scope**(**bind**("x", 7), **eval**(
ast-term("bound", **ast-value**(**identifiers**, "x")))))
→ **give**(**ast-term**("bound", **ast-value**(**identifiers**, "x")), **scope**(**bind**("x", 7), **bound**("x")))
→ 7

Figure 8.2: An example of run-time evaluation of a funcon term with meta-programming.

$$\frac{T, V \Rightarrow_{ty} \mathit{true}\langle\rangle}{\mathit{ast\text{-}value}\langle T, V\rangle \Downarrow V}\ \text{(MEDIUM)} \tag{8.124}$$

$$\frac{X \Downarrow X' \qquad Xp \Downarrow Xp'}{X, Xp \Downarrow X', Xp'}\ \text{(MEDIUM)} \tag{8.125}$$

The following rules for $\Uparrow$ convert terms into their meta-representation:

$$\frac{\texttt{term-constructor}(X) \rhd C \quad \texttt{term-arguments}(X) \rhd Xp \quad Xp \Uparrow Xp'}{X \Uparrow \mathit{ast\text{-}term}(C, Xp')}\ \text{(MEDIUM)} \tag{8.126}$$

$$\frac{\texttt{term-constructor}(X) \rhd C \quad \texttt{term-arguments}(X) \rhd \#}{X \Uparrow \mathit{ast\text{-}term}(C)}\ \text{(MEDIUM)} \tag{8.127}$$

$$\frac{\texttt{values} \rhd T}{V \Uparrow \mathit{ast\text{-}value}(T, V)}\ \text{(MEDIUM)} \tag{8.128}$$

$$\frac{X \Uparrow X' \qquad Xp \Uparrow Xp'}{X, Xp \Uparrow X', Xp'}\ \text{(MEDIUM)} \tag{8.129}$$

8.6.2 Runtime HGMP

The funcon **code** takes a term x and rewrites to the AST representation of x:

$$\frac{X \Uparrow A}{\mathit{code}(X) \rightsquigarrow A}\ \text{(MEDIUM)} \tag{8.130}$$

The funcon **eval** is given an AST a and rewrites to the term represented by a.

$$\frac{asts, A \Rightarrow_{ty} true\langle\rangle \qquad A \Downarrow X}{eval(A) \rightsquigarrow X}\ \text{(MEDIUM)} \tag{8.131}$$

$$\frac{X \longrightarrow X'}{eval(X) \longrightarrow eval(X')}\ \text{(MEDIUM)} \tag{8.132}$$

As an example of runtime HGMP, consider the evaluation[6] in Figure 8.2.

8.6.3 Compiletime HGMP

The funcons in beta-release do not have compiletime semantics. Future work is required to determine how the definition of $\Rightarrow$ in [Van Binsbergen, 2018] may be integrated in a compiletime semantics for funcons. The $\Rightarrow$-relation captures a compilation phase during which several static computations can be performed, such as constant propagation and explicating implicit coercions. In the context of HGMP, this compilation phase 'searches' for quotations and anti-quotations, known as *upML* and *downML* in [Berger et al., 2017], to compute ASTs and code and insert them into the runtime version of the program. Since funcons do not (yet) have compiletime semantics, we define the $\Rightarrow$-relation to perform this search, without modifying the funcon term, as follows:

$$\frac{\texttt{term-constructor}(X) \rhd C \quad \texttt{term-arguments}(X) \rhd Xp \quad Xp \Rightarrow Xp'}{X \Rightarrow \texttt{term-construct}(C, Xp')}\ \text{(MEDIUM)} \tag{8.133}$$

$$\frac{\texttt{term-arguments}(X) \rhd \#}{X \Rightarrow X}\ \text{(MEDIUM)} \tag{8.134}$$

$$\frac{\textbf{is-terminal}(\longrightarrow, V)}{V \Rightarrow V}\ \text{(MEDIUM)} \tag{8.135}$$

$$\frac{X \Rightarrow X' \qquad Xp \Rightarrow Xp'}{X, Xp \Rightarrow X', Xp'}\ \text{(MEDIUM)} \tag{8.136}$$

By introducing the $\Rightarrow$-relation we can distinguish between two stages of evaluation. Compiletime evaluation is modelled by a single transition in the $\Rightarrow$-relation and runtime evaluation by a sequence of transitions in the $\longrightarrow$-relation. If the rules above were the only rules to define $\Rightarrow$, then compilation has no effect ($\Rightarrow$ is an identity function). Below we shall see that rules with a higher priority are given to define the compiletime semantics of quotations and

[6] Rewrite steps have been omitted.

anti-quotations. Rule (8.133) expresses that if X is not a value[7], its subterms are compiled and possibly replaced. Rule (8.135) determines that values are not changed by compilation, even when they have computations as subterms, e.g. $\mathit{abstraction}\langle \mathit{fail} \rangle \Rightarrow \mathit{abstraction}\langle \mathit{fail} \rangle$.

Funcons **meta-up** and **meta-down** correspond respectively to the *upML* and *downML* from [Berger et al., 2017], and are the compiletime version of **code** and **eval**:

$$\frac{T \Uparrow A}{\mathit{meta\text{-}up}(T) \Rightarrow A}\ (\text{HIGHER}) \qquad (8.137) \qquad \frac{X_0 \Uparrow X_1 \quad X_1 \Uparrow X_2}{\mathit{meta\text{-}up}(X_0) \Uparrow X_2}\ (\text{HIGHER}) \qquad (8.138)$$

$$\frac{X_0 \Rightarrow X_1 \quad X_1 \dashrightarrow A \quad A \Downarrow X_2}{\mathit{meta\text{-}down}(X_0) \Rightarrow X_2}\ (\text{HIGHER}) \qquad (8.139) \qquad \frac{X \Rightarrow X'}{\mathit{meta\text{-}down}(X) \Uparrow X'}\ (\text{HIGHER}) \qquad (8.140)$$

The rules have a higher priority to overrule the 'default behaviour' of $\Rightarrow$ and $\Uparrow$ expressed in Rules (8.133) and (8.126). The funcon **meta-down** triggers runtime evaluation at compiletime. At compiletime, **meta-down**(x_0) is replaced by x_2 if x compiles and evaluates to an AST a with $a \Downarrow x_2$.

Rule (8.140) shows that an occurrence of **meta-down** within an occurrence of **meta-up** is 'cancelled out', resulting in a partially evaluated AST. For example, consider the computation **meta-up**(**give**(3, **meta-down**(**bound**(`"x"`)))), which compiles to the term $t =$ **ast-term**(`"give"`, **ast-value**(**naturals**, 3), **bound**(`"x"`)). If t occurs in a context in which `"x"` is bound to an AST, then t evaluates to an AST. In this example, the computation **eval**(**scope**(**bind**(`"x"`, **code**(**given**)), t)) evaluates to 3.

In this example, `"x"` is bound at runtime. To bind identifiers at compiletime, we introduce **meta-let**, corresponding to *letdownML* [Berger et al., 2017]. It makes (non-local) bindings available, at compiletime, to occurrences of **meta-down**, which is necessary to write

[7] The CBS to IML translation is such that there are no values of the form $f(x_1, \ldots, x_n)$.

non-trivial HGMP programs:

$$\frac{X_1 \Rightarrow X_1' \quad X_1' \dashrightarrow I \qquad X_2 \Rightarrow X_2' \quad X_2' \dashrightarrow V \qquad map(tuple(I,V)) \rightsquigarrow^* Rho_2 \qquad \texttt{map-override}(Rho_1, Rho_2) \rhd Rho_3 \qquad X_3, \texttt{environment} = Rho_3 \Rightarrow X_3'}{meta\text{-}let(X_1, X_2, X_3), \texttt{environment} = Rho_1 \Rightarrow X_3'} \text{(HIGHER)} \tag{8.141}$$

The first argument is compiled and evaluated to an identifier i. The second argument is compiled and evaluated to a value v. The binding $i \mapsto v$ is active in the compilation of the third argument x_3 to x_3', which replaces $metalet(x_1, x_2, x_3)$ during compilation.

The funcons **meta-down**, **meta-up**, and **meta-let** have no runtime semantics as their occurrences are removed at compiletime.

8.7 Static Refocussing

In this section we use the CBS to IML translation to reason about the correctness of applying *refocussing* to funcon definitions. Refocussing was introduced as an alternative evaluation strategy in the context of reduction semantics by [Danvy and Nielsen, 2004]. Danvy and Nielsen proved the strategy to be sound and indeed more efficient than 'straightforward' interpretation. We use the term *dynamic refocussing* to refer to this kind of refocussing, implemented as an optimisation, or alternative evaluation strategy, in an interpreter.

In [Danvy and Nielsen, 2004], the format in which transition (reduction) steps are specified is significantly more restricted than the format of CBS rules. Most importantly, compared to CBS, there is no notion equivalent to that of auxiliary semantic entity. In his thesis, Bach Poulsen introduces a particular form of MSOS rules, and proves that refocussing is valid when applied to specifications consisting only of rules in this form [Bach Poulsen, 2016]. Moreover, Bach Poulsen describes a mechanical transformation on small-step inference rules that encodes refocussing directly into the rules [Bach Poulsen and Mosses, 2014a]. This means that it is not necessary to implement refocussing as part of an interpreter. We refer to this transformation as *static refocussing*. In this section we implement static refocussing as part of the CBS to IML translation.

All of the funcons in beta-release are defined with rules referring to at most one semantic

entity, and all funcons, except **atomic**, are defined with rules that have at most one $\longrightarrow$-premise. Translating the rules with a single premise, we get IML rules of the form:

$$\frac{\ldots \qquad X_i \rightsquigarrow^* Y_i \qquad Y_i, \mathsf{e} = E_2 \longrightarrow X_i', \mathsf{e} = E_2' \qquad \ldots}{f(X_1, \ldots, X_i, \ldots, X_n), \mathsf{e} = E_1 \longrightarrow f(X_1, \ldots, X_i', \ldots, X_n), \mathsf{e}_1 = E_1'} \text{ (MEDIUM)} \tag{8.142}$$

(In the inference rules above, the ellipsis above the bars stand for arbitrary conditions other than $\longrightarrow$-transitions.) In what follows, we refer to rules of the form (8.142) as congruence rules. Adopting the terminology of [Danvy and Nielsen, 2004], if a congruence rule is a applicable to a configuration with program term t, then the rule *decomposes* t by pattern matching it against the source of the rule's conclusion. The target of the rule *composes* a term t' which differs only from t in that one immediate subterm g is replaced as the result of executing the rule's premise. If the premise is established via the application of an axiom, we refer to g as a redex. Otherwise, g is decomposed so that one of its immediate subterms is a redex or further decomposed, etc. This inductive process of finding a redex within a term t is *decomposition*. The process of replacing the redex and composing a term otherwise identical to t is *recomposition*.

The idea behind refocusing is that, if a term t_1 transitions to t_2 via decomposition and recomposition with redex g_1, replaced by g_2, then subsequent decomposition of t_2 would select g_2 as the next redex to be replaced, unless g_2 is terminal. Instead of replacing g_1 with g_2, we can therefore replace g_1 with g_n instead, if there is a computation of length at least one from g_1 to g_n. By doing so, a potentially large amount of redundant decomposition and recomposition steps can be avoided. There are at least two properties that a specification must have to be sure that t_2 will be decomposed by the same congruence rules that decomposed t_1. Firstly, the specification must be deterministic, guaranteeing that the same congruence rules will be applied if they are still applicable. Secondly, the congruence rules must still be applicable, which may not be the case when the values of semantic entities changed. Our experiments with static (Section 8.8) and dynamic (§13.2.2) refocussing have not revealed funcons for which these properties do not hold.

Premise translation Static refocussing is implemented by changing the way in which $\longrightarrow$-premises are translated to IML. If, without static refocussing, a premise of the form $X_0, E_1 = X_1, \ldots, E_n = X_n \longrightarrow Y_0, E'_1 = Y_1, \ldots, E'_m = Y_m$ is generated for a particular CBS rule, then with static refocussing, the following premises are generated instead (with X'_0 a fresh variable and all variables X'_0, X_i and Y_i sequence variables):

$$X_0, E_1 = X_1, \ldots, E_n = X_n \longrightarrow X'_0$$
$$X'_0 \dashrightarrow Y_0, E'_1 = Y_1, \ldots, E'_m = Y_m$$

Operationally, our implementation of static refocussing is such that, in order to establish a premise, one or more transitions are performed until the result is a value sequence, instead of exactly one transition after which the result does not have to be a value sequence. The premise is established if the resulting value sequence matches the pattern of the premise. Any changes to the values held by entities are propagated between transitions.

For example, the following rule is generated with static refocussing for **scope** (as a replacement for Rule (8.95)):

$$\frac{\begin{array}{c} \mathit{environments}, \mathit{Rho}_1 \overset{\rightsquigarrow}{\Rightarrow}_{ty} \mathit{true}\langle\rangle \quad \mathit{map\text{-}override}(\mathit{Rho}_1, \mathit{Rho}_2) \rightsquigarrow^* Y \\ X \rightsquigarrow^* Z \quad Z, \texttt{environment} = Y \longrightarrow X'_0 \quad X'_0 \dashrightarrow X' \end{array}}{\mathit{scope}(\mathit{Rho}_1, X), \texttt{environment} = \mathit{Rho}_0 \longrightarrow \mathit{scope}(\mathit{Rho}_1, X'), \texttt{environment} = \mathit{Rho}_0} \text{(MEDIUM)} \tag{8.143}$$

Remarks on correctness Consider the IML specifications generated without and without static refocussing from a particular collection of funcon definitions. Proving the soundness of static refocussing involves proving that $\gamma_1 \dashrightarrow \gamma_2$ holds in one of these specifications if and only if it holds in the other. In other words, with or without refocussing, programs compute the same result and have the same side-effects according to both specifications. That static refocussing is an optimisation can be shown by proving that the derivation of $\gamma_1 \dashrightarrow \gamma_2$ in the refocussed specification is smaller (in the number of nodes) than the derivation of $\gamma_1 \dashrightarrow \gamma_2$ in the standard specification. (Because we assume that funcon definitions are deterministic there is only one derivation of $\gamma_1 \dashrightarrow \gamma_2$ in either specification.)

8.8 Evaluation

By translating funcon definitions to IML, we have given an execution model to CBS rules and obtained a method for executing funcon terms. This translation has been implemented in the CBS compiler (package `funcons-intgen`). IML specifications have been generated from most CBS files defining the funcons in beta-release. The generated specifications are combined with IML files containing foundational declarations and rules such as those given in Sections 8.1 and 8.6, files that implement built-in funcons and operations, and files that contain queries. In total, there are over 460 queries confirming the behaviour of funcons. Of the 460+ queries, 109 larger queries have been generated from tests developed by Peter Mosses (written as configuration files, see Section 9.4). These queries are executed by giving the IML interpreter (package `iml-tools`) all the required IML files. The IML specifications, with and without static refocussing, are provided as supplementary material.

The queries execute successfully, with the expected output, both with and without static refocussing. With static refocussing, parsing the IML files took 23 seconds. Subsequently executing all queries, and printing derivations to output files, took another 4 minutes and 53 seconds. Without refocusing, parsing took 22 seconds, and executing queries and printing derivations took 172 minutes and 31 seconds. This experiment ran on a laptop with quad-core 2.4GHz processors and 8GiB of RAM, under Ubuntu 14.04.

Chapter 9

The Haskell Funcon Framework

References and Acknowledgements *The material in this chapter has been presented at the 27th Nordic Workshop on Programming Theory (NWPT 2015) and at the 15th International Conference on Modularity (MODULARITY 2016) [Van Binsbergen et al., 2016] and is derived from draft versions of [Van Binsbergen et al., 2019]. The Haskell code of this chapter is joint work with Neil Sculthorpe.*

Chapter 8 introduced funcons and CBS funcon definitions. By translating funcon definitions to IML, we gave an execution model to CBS rules and obtained a method for executing funcon terms.

This chapter discusses a second method for executing funcon terms: the Haskell Funcon Framework (package `funcons-tools`), a specialised set of tools for defining and executing funcons in Haskell. The framework has been used to verify the CBS specifications of example languages by executing programs designed to test specific aspects of the language. Section 13.2 provides experimental data to support the claim that the framework is suitable for this purpose.

In the framework, a so-called *micro-interpreter* is defined for each funcon. The micro-interpreters of arbitrary collections of funcons are freely combined. A funcon term interpreter for a language specified within CBS is obtained by combining the micro-interpreters of the funcons used in the language's specification. The CBS compiler can generate micro-interpreters from CBS funcon definitions.

This chapter discusses some of the technical aspects of micro-interpreters, focussing on their modularity. This chapter also explains how ambiguous patterns and type sequences are handled, configuration files for defining implementation-specific funcons and unit-tests, as well as the dynamic refocussing optimisation.

9.1 Funcon Modules

The Haskell Funcon Framework (package `funcons-tools`) contains the following modules:

- `Funcons.EDSL` exports helper functions that are used in the implementation of funcons. For example, to modify or access semantic entities, or to test side-conditions.
- `Funcons.Tools` exports helper functions for composing *funcon modules*. A funcon module is a Haskell module exporting three collections containing information about funcon, data type, and semantic-entity definitions, respectively. Each funcon module exports an interpreter, which is aware only of the components defined in the module.
- `Funcons.Core`, a funcon module containing the implementation of the funcons in the reusable library. This includes all the funcons mentioned in this thesis, except the funcons for HGMP, which are implemented by `Funcons.MetaProgramming` (see §9.5).

Funcon modules have two noteworthy properties. Firstly, they are independent and do not need to import other funcon modules; they only need to import `Funcons.EDSL`, `Funcons.Operations` (see §7.4), and `Funcons.Tools`. Secondly, funcon modules can be freely composed: a module implementing funcons A and a module implementing funcons B are composed without restrictions to form a module implementing funcons A and B. CBS files are thus compiled individually, and the resulting modules can be composed as needed. Moreover, once a module has been compiled, it can be linked as needed, without requiring recompilation. This makes it possible to deliver the implementation of the reusable funcon library as an isolated package. Enforcing these properties has been a leading principle in the development of the Haskell funcon framework. For the purpose of illustration we have compiled a contrived CBS file containing the definitions of **environments**, **scope**, and **returning**. Figure 9.1 shows a fragment of the funcon module generated from this file. It omits the functions that implement the behaviour of the funcons, e.g. *stepEnvironments*.

```
import Funcons.EDSL
import Funcons.Operations hiding (Values, libFromList)
import Funcons.Tools
main = mkMainWithLibraryEntitiesTypes funcons entities types
entities = []
types = typeEnvFromList
  [("returning", DataTypeMembers "returning" []
    [DataTypeConstructor "returned"
      [TName "defined-values"] (Just [])])]
funcons = libFromList
  [("environments", NullaryFuncon stepEnvironments)
  ,("scope", NonStrictFuncon stepScope)
  ,("returning", NullaryFuncon stepReturning)
  ,("returned", StrictFuncon stepReturned)]
```

Figure 9.1: Excerpt of a funcon module for a contrived collection of CBS definitions.

We represent funcon terms using strings for the names of funcons in a data type, with generic constructors for funcon application, constants (funcons with no arguments), and literals (built-in values and types). A funcon module contains a *funcon library* mapping funcon names to their implementation (*funcons* in Figure 9.1). Similarly, we use maps to implement entity records, associating the names of entities with the values they hold. Evaluating funcon terms can cause runtime errors because it involves searching in maps and there are no static guarantees that the required entries are available. By using strings in these ways we have gained the desired modularity, but lost static guarantees provided by a Haskell compiler.

An alternative would have been the *Data types à la carte* technique by [Swierstra, 2008], combining funcons defined in separate modules into a single data type with a constructor for each funcon. Individual Haskell constructors for each funcon would provide stronger correct-by-construction guarantees about the well-formedness of funcon terms. Several authors [Day and Hutton, 2012, Wu et al., 2014] have used this technique for the specific purpose of defining modular programming-language constructs in a Haskell setting.

Types are either built-in to CBS (e.g. **integers** and **values**), provided as reusable components (e.g. **environments** and **identifiers**), or defined as part of a language specification. A funcon module generated from a CBS file provides a funcon implementation for each type

and data-type constructor defined in the file (see *funcons* in Figure 9.1). In addition, a funcon module provides a type environment which relates the data types to their constructors (*types* in Figure 9.1) and stores information about parameters. Funcon term interpreters use the type environment for dynamic type checking.

9.2 CBS Rules

In Section 8.4 we described how most classes of semantic entities are implicitly propagated in CBS rules. We achieve implicit propagation the Haskell Funcon Framework by working in a monad, and defining its *return* and *bind* operations such that they implement the desired implicit propagation. A contextual entity corresponds to a reader monad, a mutable entity corresponds to a state monad, and an output entity corresponds to a writer monad [Jones, 1995]. Input entities correspond to a restricted form of state monad (input is always consumed in order, and each input can only be consumed once), and control-flow entities correspond to a combination of a reader and a writer monad (except that only one signal may be emitted in each control-flow entity at once; they do not form a monoid).

We define a data type *MSOS* that combines the five semantic-entity classes, omitting the monad instance:

$$
\begin{aligned}
&\textbf{data}\ \mathit{MSOS}\ a \ = \mathit{MSOS}\ (\forall_m.\ \mathit{Interactive}\ m \Rightarrow \\
&\qquad\qquad \mathit{MReader}\ m \to \mathit{MState}\ m \to \\
&\qquad\qquad\quad m\ (\mathit{Either}\ \mathit{Exception}\ a, \mathit{MState}\ m, \mathit{MWriter})) \\
&\textbf{data}\ \mathit{MReader} \ = \mathit{MReader}\ \mathit{Contextual}\ \mathit{ControlFlow}\ (\mathit{InputDefault}\ m) \\
&\textbf{data}\ \mathit{MState}\ m = \mathit{MState}\ \mathit{Mutable}\ (\mathit{Input}\ m) \\
&\textbf{data}\ \mathit{MWriter} \ = \mathit{MWriter}\ \mathit{ControlFlow}\ \mathit{Output} \\
&\textbf{data}\ \mathit{Exception} = \ ...
\end{aligned}
$$

The type *Contextual* is the type of mappings from contextual entity names to values (elements of *Values*, import from `Funcons.Operations`), closely resembling an entity record. The type *Mutable* is the type of mappings from mutable entity names to values and *Output* from output entity names to lists of values. As such, we have avoided the need to use monad transformers [Liang et al., 1995] to add additional entities: our *MSOS* monad is fixed. Using Haskell's native support for monads, we can generate human-readable funcon code that only refers to the entities explicitly mentioned in the funcon definition. As a consequence, the code of a funcon is as modular and compositional as its CBS definition. Moreover, the

code is only recompiled if the CBS file in which the funcon is defined is regenerated. The code for a funcon is formed by a sequence of statements in the *MSOS* monad for every rule of the funcon. The statements can access or modify semantic-entity values, perform pattern matching and substitution, and test side conditions and premises. The statements generated for a rule may raise several kinds of internal exceptions, each representing a particular reason why the rule implemented by those statements is not applicable. For example, a funcon can be applied to incorrect arguments (e.g. if the first argument of **if-true-else** was not a Boolean), pattern matching may have failed, or a premise may not hold. To handle failure, the *MSOS* monad contains the *Exception* component and backtracking is used to try an alternative rule. If no rules are applicable to a particular term, i.e. all alternatives raise an exception, then the term is stuck.

The *MSOS* monad stores extra information, including the funcon library, runtime options, and collected meta-data; but we elide those details here. The code for a funcon forms a micro-interpreter that can be executed independently, given a funcon library with entries for all funcons explicitly mentioned in the definition of the funcon. As an alternative to input/output simulation, the *MSOS* monad also provides facilities for connecting input and output entities to real console input/output, allowing a user to run a funcon program interactively. Interactive input is enabled by the *Interactive* monad m appearing in the definition of *MSOS*.

9.2.1 Context-free Rewriting

The rewrite relation $\rightsquigarrow$ is context insensitive; rewrites do not modify semantic entities. To separate rewriting from computation steps we use a separate monad (not shown), which does not provide access to the semantic entities. This guarantees, by construction, that rewrites do not access semantic entities.

The properties of the rewrite relation are such that, operationally, it can be applied at any time during execution anywhere in a term. Thus, funcon term interpreters are free to perform rewriting whenever is most convenient or efficient. Experience has shown that the choice of strategy to determine when rewrites are performed has a tremendous influence on the efficiency of funcon interpreters. A greedy rewriting strategy performs rewrites immediately on every occurrence of a term, possibly resulting in redundant rewrites (e.g.

rewriting all arguments **if-true-else**). We choose to rewrite a term just before performing a computational step, and directly afterwards on the term resulting from the step, if the step was successful. This strategy exhibits inefficiencies in combination with our backtracking approach. A rule may turn out to be inapplicable after some rewrites have been performed. The rewrites are forgotten when backtracking selects the next rule to try. We expect this problem is easy to avoid in a host language providing mutable references or objects. Our Haskell implementation would benefit from graph reduction for sharing the effects of rewrites [Peyton Jones, 1987].

9.2.2 Dynamic Refocussing

Directly implementing a small-step operational semantics as the transitive closure of the computation-step relation is straightforward but inefficient. At each computation step, the interpreter traverses the funcon term from the root to a subterm (sometimes called the redex), 'executes' the subterm by replacing it, and reconstructs a (mostly identical) term (as discussed in more detail in Section 8.7). Simultaneously, the values held by semantic entities are maintained and possibly modified. This corresponds to constructing a derivation tree in accordance to the I-MSOS rules (including the implicit congruence rules) of a CBS language definition. The cost of each step is potentially linear in the size of the funcon term. The refocussing optimisation, introduced by [Danvy and Nielsen, 2004], and discussed in Section 8.7, overcomes this inefficiency.

Section 8.7 shows how refocussing can be implemented statically as part of the translation from CBS rules to IML. We implemented refocussing in the funcon term interpreters of the Haskell Funcon Framework as follows. When a redex r has been discovered, repeatedly apply transition r as long as no signal is emitted. The resulting term r' replaces r, which is a value unless a signal has been emitted. When a signal was emitted, the redex should not be further transitioned, as the signal needs to be handled further up the term. Typically, the handler for a control-flow signal will then change the subterm that is next to be executed (e.g. Rule (8.105) for **handle-abrupt**). The static refocussing implementation of Section 8.7 does not give a special treatment to signals, as signals cause termination in Chapter 8. In §13.2.2 we demonstrate the significant effects of dynamic refocussing on running times.

9.3 Ambiguous Patterns and Types

In Chapter 8, two assumptions were made about funcon definitions in CBS: a single pattern sequence does not contain multiple sequence variables and a type operator is only applied to types with only single values as members (not sequences of values). Patterns and types that violate these assumptions are ambiguous, i.e. a term sequence can match such a pattern sequence in more than one way, and a value sequence can be shown to be a member of such a type in more than one way. The first assumption does not hold for all funcons currently in beta-release, most importantly **left-to-right**, as discussed in Section 8.5. In Section 8.5 we also saw an example of an ambiguous type, in relation to **environments**, for which straightforward disambiguation did not suffice. In this section, we summarise the pattern matching algorithm of the Haskell Funcon Framework to show how funcon term interpreters deal with such patterns. We also demonstrate that the pattern matching algorithm can used for type checking with ambiguous types. The code given in this section is simultaneously a simplification and a generalisation of the actual implementation, used for illustrative purposes.

9.3.1 Pattern Matching with Sequence Variables

CBS patterns are regular patterns — as we find them in many functional programming languages — with two optional forms of annotation. Variables can be annotated with a type, e.g. X : **booleans**, specifying that X can only be bound to a value of type **booleans**. Variables can also have a sequence annotation in the form of a superscript, e.g. X^*, X^+, or $X^?$, specifying that X can be bound to 'zero or more', 'one or more', and 'zero or one' terms respectively. Both forms of annotations can be combined, e.g. X^+ : **values***. The patterns of the XML-centric functional language ℂDuce allow similar annotations on variables (and patterns generally) [Benzaken et al., 2003].

Simple patterns The pattern matching algorithm of this section uses Wadler's list-of-successes method [Wadler, 1985] to capture the possibly many results of matching against an ambiguous pattern. The core concept is a *Matcher*, a function given a term sequence (type [*Term*]) and a set of current bindings (*Bindings*) returning a list of successful matches. Each

match is a pair $([\,Term\,], Bindings)$ containing an extended set of bindings and the terms 'consumed' by the matcher, a prefix of the input sequence.

$$\textbf{type}\ Matcher = [\,Term\,] \to Bindings \to [([\,Term\,], Bindings)]$$

The type *Matcher* is similar to the type *Parser* of parser combinator expressions in Section 5.2, and indeed, Wadler used parsing and pattern matching as examples of his list-of-successes method [Wadler, 1985]. Wadler also suggested that his pattern matching algorithm may be generalised and used for different purposes. Our extension with type checking and type annotations is an example. We proceed by giving 'elementary matchers' and 'matcher combinators', reminiscent of Rhiger's Functional Pearl [Rhiger, 2009]. Compared to [Rhiger, 2009], our patterns are untyped, making it possible to attempt to match a term and a pattern of different structures, although these attempts will certainly fail. With typed patterns, it is possible to disallow such matching attempts statically. On the other hand, Rhiger does not discuss patterns that match sequences of terms. Future work is required to determine how sequence-matching patterns can be typed.

The first matcher we define is a matcher for a pattern consisting of just a variable (*Var*):

```
match_var :: Var → Matcher
match_var x [] env       = []                          -- no match, nothing to be consumed
match_var x (t : ts) env = [([t], bind x [t] env)]     -- single match, of length 1
bind :: Var → [Term] → Bindings → Bindings             -- adds a binding to a set of bindings
```

A sequence variable can consume all prefixes of a term sequence:

```
match_seq_var :: Var → Matcher
match_seq_var x ts env = map toRes (inits ts)   -- inits from Data.List gives all prefixes
  where toRes prefix = (prefix, bind x prefix env)
```

Types and annotations A type is a function from a term sequence to a Boolean so that a term sequence *ts* is a member of the type *ty* when $ty\ ts \equiv True$.

$$\textbf{type}\ Type = [\,Term\,] \to Bool$$

Type annotations are implemented by a higher-order function that constructs a matcher given a matcher and a type. The function runs the given matcher and filters its results to accept only those results for which it holds that the consumed term sequence is a member of the given type.

```
match_ann :: Matcher → Type → Matcher
match_ann p ty ts env = filter (ty ∘ fst) (p ts env)
```

When applied to a sequence variable, sequence operators can be seen as a types. For example, as a type, the **?** operator contains all term sequences of length zero or one:

```
starOp, plusOp, qmOp :: Type
starOp ts = True
plusOp ts = length ts ⩾ 1
qmOp ts   = length ts ⩽ 1
```

The pattern X^+ is then implemented as follows, using strings as variables:

```
example1 = match_ann (match_seq_var "X") plusOp
```

Disambiguation A matcher is ambiguous if there are term sequences which it matches in more than one way, i.e. term sequences for which it returns a list containing more than one result. Reducing the ambiguity of a matcher is filtering the result lists it produces, so annotations can be seen as ambiguity reduction operators. Disambiguating a matcher means reducing its ambiguity such that it returns exactly one result (or no result if the original matcher gives no result). We let the order of a result list determine a preference between results, in descending order, so that the least preferred result is at the end of the list. Disambiguation is then simply selecting the first result of a result list, if there is any. The functions *match_longest* and *match_shortest* sort the result list of a given matcher, so that the results that consume longer respectively shorter term sequences are preferred.

```
match_sort :: (Int → Int → Ordering) → Matcher → Matcher
match_sort sorter p ts env = sortBy (sorter `on` (length ∘ fst)) (p ts env)
match_longest, match_shortest :: Matcher → Matcher
match_shortest = match_sort compare          -- sortBy compare gives ascending lists
match_longest  = match_sort (flip compare)   -- sortBy (flip compare) gives descending lists
```

Function *matches* attempts to match a term sequence against a pattern sequence:

```
matches :: [Term] → [Matcher] → Bindings → Maybe Bindings
matches ts ps env = case foldr match_seq match_final ps ts env of
                      []          → Nothing
                      ((_, env): _) → Just env   -- disambiguation
  where
    match_seq :: Matcher → Matcher → Matcher
    match_seq p q ts env = try (p ts env)
      where try [] = []
```

```
  try ((ts', env') : rest) = case q (drop (length ts') ts) env' of
    []    → try rest                                          -- backtracking
    res   → [(ts' ++ ts'', env'') | (ts'', env'') ← res]     -- disambiguation
match_final :: Matcher
match_final [] env = [([], env)]
match_final _ _    = []
```

The combinator *match_seq* combines matchers, performing left-biased disambiguation, i.e. *match_seq p q* considers the preferences of p before the preferences of q. A term sequence is only matched if the pattern sequence consumes all terms. This follows from the definition of the *match_final* matcher, which only succeeds if it is given the empty sequence.

When *match_longest* and *match_shortest* are applied to sequence variables (applications of *match_seq_var*) we obtain matchers that behave similarly to the greedy and ungreedy sequence capture variables of ℂDuce [Benzaken et al., 2003] respectively.

Patterns depending on bindings None of the matchers we have given so far actually inspects the given set of bindings. However, by propagating the bindings this way, it is possible to implement patterns that perform substitution. For example, the following alternative definition of *match_var* can be useful when implementing patterns in which a variable can occur multiple times.

```
match_var' :: Var → Matcher
match_var' x [] env = []
match_var' x (t : ts) env = case bound x env of    -- is there a binding for x?
  Nothing          → [([t], bind x [t] env)]       -- no, behave as match_var
  Just [t'] | t ≡ t' → [([t], env)]                -- yes, then succeed if equal to [t]
  _                → []                            -- and fail otherwise
```

9.3.2 Type Checking with Ambiguous Types

In the discussion so far, we skipped over the implementations of built-in and user-defined CBS types and have determined that type checking is simply applying a type (as a function) to a term sequence to detect membership. In the examples that follow we use the following simple types:

```
ty_values, ty_booleans, ty_integers :: Type
ty_values   = ...
ty_booleans = ...
ty_integers = ...
```

Type operators (see §8.1.2) such as type union |, and type repetition with ∧ and **+**, are implemented as higher-order functions over types. For example, type-complement (type-negation) is implemented as follows:

```
ty_neg ty ts = not (ty ts)
```

As another example, applied to a type t, the operator **?** returns a type which contains the empty sequence and all members of type t.

```
ty_opt :: Type → Type
ty_opt ty = ty_union null ty
ty_union :: Type → Type → Type
ty_union ty1 ty2 ts = ty1 ts ∨ ty2 ts
ty_inter :: Type → Type → Type
ty_inter ty1 ty2 ts = ty1 ts ∧ ty2 ts
```

A type sequence may consist of types that contain the empty sequence or sequences of length greater than one. The IML Rules (8.19) and (8.20) given in §8.1.2 are problematic because they assume each type in a type sequence has only single terms as members. The following implementation of type sequences corresponds closely to these problematic rules:

```
ty_seq' :: [Type] → Type
ty_seq' tys ts = and $ zipWith provide tys ts
  where provide ty t = ty [t]
```

This definition checks each term and type in the given sequences pairwise. As a result, if the singleton term sequence **true** is checked with type sequence (**integers**?, **booleans**), then **true** is checked to be of type **integers**?, which fails, and the type **booleans** is ignored.

Alternatively, we take advantage of the pattern matching machinery to implement type sequences (*noBindings* is the empty set of bindings):

```
ty_seq :: [Type] → Type
ty_seq tys ts = isJust (matches ts ps noBindings)
  where ps = zipWith toPat [1..] tys
    where toPat i ty = match_ann (match_seq_var ("X" ++ show i)) ty
ty_power :: Type → Int → Type
ty_power ty n = ty_seq (replicate n ty)
```

The main insight is: given a term sequence $t_1, \ldots, t_m$, and a type sequence $\tau_1, \ldots, \tau_n$, we can use sequence variables the find all possible length n partitionings $t_1^*, \ldots, t_n^*$ of $t_1, \ldots, t_m$ for which hold that t_i^* is a member of τ_i, for all $1 \leqslant i \leqslant n$. If at least one such a partitioning exists, then the given term sequence is a member of the give type sequence.

Also challenging are the type operators * and +. Previously we considered $Ty^?$, for any type Ty, as equivalent to # | Ty, where # is the type of the empty sequence. Similarly, we can view Ty^* as equivalent to # | Ty | (Ty, Ty) | (Ty, Ty, Ty) | ... of which a direct implementation would result in nontermination, as there are infinitely many alternatives to try. Since there are infinitely many alternatives, compared to type sequences, we have to find n together with a length n partitioning of the given term sequence. As shown by the code below, we do so by recursively consuming prefixes of the given term sequence using pattern matching, guaranteeing that each consumed prefix is a member of Ty. This is done until all terms have been consumed. If n recursive calls consume all input terms, then a length n partitioning exists.

```
ty_star :: Type → Type
ty_star ty [] = True
ty_star ty ts = select $ matcher ts noBindings
  where matcher = match_longest (match_ann (match_seq_var "X") ty)
        select []                                = False
        select (([], _): _)                      = False
        select ((ts', _) : rest)
          | ty_star ty (drop (length ts') ts) = True
          | otherwise                         = select rest   -- backtracking
```

Backtracking ensures that all possible partitionings are tried. Nontermination is a risk if the type given to *ty_star* contains the empty sequence. To avoid nontermination, we use *match_longest* in the definition, ensuring that no input is consumed only after all prefixes have been tried. We know that type checking fails in this case, because the ordering tells us that the given type only matches the empty prefix of the current input, no further attempts will consume any input, and the current input is not empty.

The implementation of + is in terms of *ty_star*:

```
ty_plus :: Type → Type
ty_plus ty ts = select $ matcher ts noBindings
  where matcher = match_longest (match_ann (match_seq_var "X") ty)
        select []                                = False
        select ((ts', _) : rest)
          | ty_star ty (drop (length ts') ts) = True
          | otherwise                         = select rest
```

There are two differences in the definition of *ty_plus* compared to *ty_star*. Firstly, a partitioning of the input must be found that is of at least length one, corresponding to the interpretation of Ty^+ as Ty | (Ty, Ty) | (Ty, Ty, Ty) | ... (which may still consume no

terms, as shown by the type *ty_plus* (*ty_opt ty_booleans*) that contains the empty sequence). Secondly, there is no risk of nontermination as *ty_star* is not recursive (*select* is recursive, but there is only a finite amount of prefixes to consider.)

We now have all the necessary implementations to translate arbitrary CBS patterns. Only sequence variables can match more than one term and require disambiguation. The chosen strategy is to match sequence variables with annotation greedily and those without annotations lazily. The patterns X^+ : **booleans*** and $Y^?$ are thus translated to:

```
example2 = match_longest $
               match_ann (match_ann (match_seq_var "X") plusOp) (ty_star (ty_booleans))
example3 = match_shortest $ match_ann (match_seq_var "Y") qmOp
```

9.4 Configuration Files

Funcon term interpreters are easily configured by command line flags or configuration files. For example, a user can restrict the number of computational steps performed. This is useful to determine which steps lead up to unwanted behaviour (arising from an invalid program or an incorrect specification). The interpreters can print debugging information such as the values of semantic entities, and meta-data such as the number of computational steps performed and the number of rewrites performed. The documentation of `Funcons.Tools` contains an exhaustive enumeration of all configuration options.

Configuration files offer a limited method for defining funcons. A particular use case is to define *implementation-specific* funcons representing a particular implementation choice. Language manuals often leave room for language implementations regarding certain aspects of the language. For example, the number of bits used in the binary representation of different forms of integers. The following excerpt shows the definition of a number of implementation-specific funcons in a configuration file:

```
funcons {
    implemented-floats-format   = binary64;
    implemented-integers-width  = 31;
    implemented-characters      = unicode-characters;
}
```

Configuration files also offer a simple but effective method for defining unit tests. A configuration file for a unit test contains an initial term to be evaluated and the expected

outcome of evaluating the term. The expected outcome consists of a result term, as well as the values of some semantic entities. The following excerpt shows an example of a unit test:

```
general{
  funcon-term:  give(read,sequential(print(given),print(given)));
}
inputs{
  standard-in:  2,3;   // provides 2 and 3 as input
}
tests{
  result-term:  null;  // expects null as the result value
  standard-out: 2,2;   // expects 2 is printed twice
  standard-in:  3;     // expects 3 as remaining input
}
```

The differences between expected and actual outcome are reported to the standard output. A successful unit test is silent. Entities not mentioned in a unit test are treated differently depending on the class of the entity. For example, any value for store is considered acceptable outcome, if the tests does not mention store. However, a raised abrupted signal is considered a violation if the test did not mention abrupted.

9.5 Homogeneous Generative Meta-Programming

Module `Funcons.MetaProgramming` implements the funcons for HGMP according to their definitions in [Van Binsbergen, 2018] and similar to their implementations in Section 8.6. The module contains manual implementations of the $\Uparrow$ and $\Downarrow$ relations and exports manual implementations of **eval**, **code**, **meta-up**, **meta-down**, **meta-let**, and the compiletime relation $\Rightarrow$.

With the BNF combinators of the `gll` package, `Funcons.MetaProgramming` has been used to implement the HGMP-extended λ_{v} language of [Van Binsbergen, 2018]. As a code example of this implementation, we give the funcon translation of the most interesting operator of λ_{v}, $!x$, which enables sharing of runtime AST evaluation:

$$\begin{aligned} \mathit{sem\_splice\_id}\ x = \ & \textbf{give\_}\,[\textbf{eval\_}\,[\textbf{current\_value\_}\,[\textbf{bound\_}\,[x]]] \\ & ,\textbf{sequential\_}\,[\textbf{assign\_}\,[\textbf{bound\_}\,[x],\textbf{ast\_value\_}\,[\textbf{values\_},\textbf{given\_}]] \\ & \quad ,\textbf{given\_}]] \end{aligned}$$

The identifier x is expected to be bound to a variable which is assigned an AST (as specified by the usage of **bound_** and **current_value_**). The AST is transformed to code and evaluated

to a value v (with **eval**_). The expression $!x$ evaluates to v and assigns v, as a side-effect, to the variable to which x is bound. The value v (denoted by **given**_) is first assigned (**assign**_) and then produced as a result (last argument of **sequential**_). The result is that the value v is shared with subsequent occurrences of $!x$. The origin of funcon term constructors such as **give**_ and **eval**_ is discussed in Section 10.2.

Part III

Evaluation

Chapter 10

Tools

Parts 1 and 2 of this thesis describe techniques for defining and executing reusable components. These techniques are implemented in practical tools that are available as part of the supplementary material of this thesis. The `gll` package implements BNF combinators on top of an implementation of FUN-GLL. The `funcons-tools` package delivers the Haskell Funcon Framework with tools for constructing and executing funcon terms. In this part of the thesis, we evaluate the tools provided by these packages, examining their usability and runtime efficiency. In Chapters 11 and 12 we use these tools to formally describe two programming languages in the form of case studies. We reflect on the case studies in Section 13.3 and present empirical data on the runtime efficiency of the tools in Sections 13.1 and 13.2. In this chapter we introduce some of the relevant functions of these tools, required for the case studies.

10.1 Functions for Describing Syntax

In this section we show some of the functions exported by the `GLL.Combinators.Interface` module of the `gll` package. For a comprehensive enumeration we refer to the documentation provided with the package. The `GLL.Combinators.BinaryInterface` module exports the same functions as `GLL.Combinators.Interface`. The difference between the two modules is that the functions of `GLL.Combinators.Interface` have flexible types based on type-

classes, whereas the functions of `GLL.Combinators.BinaryInterface` coerce all combinator expressions to symbol expressions, thus performing grammar binarisation, as discussed in §5.3.4.

A syntax description consists of one or more top-level definitions of symbol expressions, which may be mutually recursive. Each of these symbol expressions describes the syntax of a language, for which the grammar can be obtained by applying the *grammarOf* function:

```
grammarOf :: (Show t, Parseable t, IsSymbExpr s) ⇒ s t a → Grammar t
parse     :: (Show t, Parseable t, IsSymbExpr s) ⇒ s t a → [t] → [a]
```

Applying *parse* to a symbol expression returns a parser for the language described by the symbol expression. Variations of *parse* exists that accept configuration options as arguments.

In §3.2.3, we discussed the need to insert a unique nonterminal name when defining a recursive, parameterised combinator so that the name is somehow based on its parameters. The function *mkNt* is introduced for this purpose:

```
mkNt :: (Show t, Ord t, IsSymbExpr s) ⇒ s t a → String → String
```

The function returns a unique nonterminal name by combining the given string (second argument) and the nonterminal name of the start symbol of the grammar generated from the given symbol expression (first argument), assuming that the given string has not been used for this purpose before. The definition of *multiple* exemplifies how *mkNt* is used:

```
multiple :: (Show t, Ord t, IsSymbExpr s) ⇒ s t a → SymbExpr t [a]
multiple p = let fresh = mkNt p "*"
             in fresh ⟨::=⟩ (:) ⟨$$⟩ p ⟨**⟩ multiple p ⟨||⟩ satisfy []
```

To explain syntax descriptions intuitively we say that a symbol expression s 'recognises' sentences of the form x as a short-hand for "the language described by s contains sentences of the form x". Similarly, we refer to the sentences of the language described by a symbol expression s as "s elements". Thus, *multiple p* recognises sequences of p elements, where p is an arbitrary symbol expression. The intention is that the nonterminal names generated by *mkNt* for *multiple x* and *multiple y* are different if x and y are different (but ideally are equal if x and y refer to the same symbol expression).

10.1.1 Tokens

`GLL.Combinators.Interface` exports certain functions that create symbol expressions, given a token of type *Token* (defined as in Section 5.1), that recognise sentences matching this token. For example:

```
int_lit     :: SubsumesToken t ⇒ SymbExpr t Int
float_lit   :: SubsumesToken t ⇒ SymbExpr t Double
id_lit      :: SubsumesToken t ⇒ SymbExpr t String              -- identifiers
alt_id_lit  :: SubsumesToken t ⇒ SymbExpr t String              -- alternative identifiers
string_lit  :: SubsumesToken t ⇒ SymbExpr t String
keyword     :: SubsumesToken t ⇒ String → SymbExpr t String     -- reserved keywords
keychar     :: SubsumesToken t ⇒ Char   → SymbExpr t Char       -- reserved characters
token       :: SubsumesToken t ⇒ String → SymbExpr t String     -- user-defined tokens
```

10.1.2 Basic Syntax

`GLL.Combinators.Interface` exports several reusable combinators, of which we have seen *multiple* as an example. A number of variatons of *multiple* exist. For example, *multiple1 p* recognises sequences of p elements of length at least one. As another example, *some p* and *many p* describe the same language as *multiple p*, but disambiguate differently. Namely, *some p* prefers a smaller number of p elements, whereas *many p* prefers a larger number. For every variation of *multiple* there is a function with a similar name extended with *SepBy* (*multipleSepBy*, *multipleSepBy1*, *manySepBy*, etc.) which receives a separator to occur between elements as an additional argument.

A more advanced variation of *multiple* does not simply return a list of semantic values, with one value for each repetition, but combines the semantic values, in a way similar to Haskell's *foldr*.

```
foldr_multiple :: (IsSymbExpr s, Parseable t) ⇒ s t (a → a) → a → BNF t a
```

The semantic value (a function) of the symbol expression argument determines how the semantic value of the rest of the sequence is extended. The second argument is a default value, taken as the semantic value of the end of the sequence.

As demonstrated by the case studies of Chapter 11 and 12, this combinator is particularly useful to describe the syntax and semantics of top-level definitions and commands. As a simpler example, consider the following combinator expression:

```
multiple_ints :: SymbExpr Token Int
multiple_ints = "ints" ⟨:=⟩ foldr_multiple odd_or_even 0
  where odd_or_even :: SymbExpr Token (Int → Int)
        odd_or_even = "odd-or-even" ⟨:=⟩ add ⟨$$⟩ int_lit
        add i total | even i      = i + total
                    | otherwise = total
```

The language described by *multiple_ints* is that of sequences of numbers. When a sentence is interpreted, the result is the sum of all the *even* numbers appearing in the sentence.

The symbol expression *optional* p recognises the sentences with a single p element and the empty sentence. The empty sentence is interpreted as *Nothing*, whereas a sentence consisting of a single p element is interpreted as *Just* v, where v is the semantic value of the element of p.

```
optional          :: (Show t, Ord t, IsSymbExpr s) ⇒ s t a       → SymbExpr t (Maybe a)
optionalWithDef :: (Show t, Ord t, IsSymbExpr s) ⇒ s t a → a → SymbExpr t a
```

The combinator *optionalWithDef* receives an additional argument, taken as the interpretation of the empty sentence instead of *Nothing*.

The combinator *within* of §3.1.4 is implemented alongside several specialisations:

```
within   :: (Show t, Ord t, IsSymbExpr s) ⇒
            SymbExpr t a → s t b → SymbExpr t c → SymbExpr t b
parens   :: (Show t, Ord t, SubsumesToken t, IsSymbExpr s) ⇒ s t b → SymbExpr t b
braces   :: (Show t, Ord t, SubsumesToken t, IsSymbExpr s) ⇒ s t b → SymbExpr t b
brackets :: (Show t, Ord t, SubsumesToken t, IsSymbExpr s) ⇒ s t b → SymbExpr t b
angles   :: (Show t, Ord t, SubsumesToken t, IsSymbExpr s) ⇒ s t b → SymbExpr t b
```

10.1.3 Expression Grammars

A common example in programming language literature is that of an expression grammar, because expressions are ubiquitous and their formal description poses several challenges. For example, grammars that capture the abstract syntax of expressions directly are highly nondeterministic and ambiguous. With the BNF combinators, the syntax of expressions can be described in this direct way. For example, consider the following description of the syntax (and semantics) of basic arithmetic expressions:

```
pExpr :: SymbExpr Token Int
pExpr = "Expr" ⟨::= ((−) ⟨$$⟩ pExpr ⟨** keyword "-" ⟨**⟩⟩ pExpr
                ⟨||⟩  (+) ⟨$$⟩ pExpr ⟨** keyword "+" ⟨⟨**⟩ pExpr)
```

```
⟨||⟩ (div ⟨$$⟩ pExpr ⟨** keyword "/" ⟨**⟩⟩⟩ pExpr
⟨||⟩  (*) ⟨$$⟩ pExpr ⟨** keyword "*" ⟨⟨**⟩ pExpr)
⟨||⟩ (int_lit ⟨||⟩ parens pExpr)
```

The usage of ⟨**⟩⟩⟩ in the alternatives for subtraction and division specifies that these operators are left-associative, whereas addition and multiplication are specified to be right-associative through the use of ⟨⟨⟨**⟩. Both these operators are variants of ⟨**⟩ performing disambiguation by preferring the longest or shortest match respectively. The alternatives are given precedence in top-to-bottom order, following the usage of ⟨::=, a variant of ⟨::=⟩, so that multiplication and division have a higher priority than addition and subtraction. Alternatives are grouped to specify that, for example, addition and subtraction have equal priority.

To simplify specifying the associativity and priorities of operators, we introduce *operator tables*, in which these aspects of operators are specified more directly.

```
type OpTable e = Map Double [(String, Fixity e)]
data Fixity e  = Prefix (String → e → e)              -- prefix operator with semantics
               | Infix (e → String → e → e) Assoc     -- infix operator with semantics
data Assoc     = LAssoc   -- left-associative
               | RAssoc   -- right-associative
               | NA       -- associativity not specified
fromOpTable :: (SubsumesToken t, Parseable t, IsSymbExpr s) ⇒
                 String → OpTable e → s t e → SymbExpr t e
```

An operator table maps priority levels of type *Double* to table entries, which determine the keyword that identifies an operator, whether the operator is a prefix or an infix operator, and, if it is an infix operator, whether it is left- or right-associative (if any). The function *fromOpTable* generates a symbol expression capturing the desired expression grammar, given a nonterminal name, an operator table, and a symbol expression. The symbol expression argument is used for the operators' operands. For example, the expression grammar defined earlier can also be defined as follows:

```
operator_syntax = opTableFromList
  [(1,    [("+", Infix sem_add  RAssoc), ("-", Infix sem_sub LAssoc)])
  ,(2,    [("*", Infix sem_mult RAssoc), ("/", Infix sem_div LAssoc)])]
  where sem_add x _ y  = x + y
        sem_sub x _ y  = x - y
        sem_mult x _ y = x * y
        sem_div x _ y  = x `div` y
pExpr' :: SymbExpr Token Int
```

$$pExpr' = \texttt{"Expr'"}\ \langle ::= \rangle\ fromOpTable\ \texttt{"operators"}\ operator\_syntax\ pExpr'$$
$$\langle || \rangle\ (int\_lit\ \langle || \rangle\ parens\ pExpr')$$

10.2 Functions for Building Funcon Terms

In this section we consider some of the functions exported by the `funcons-tools` package. For a comprehensive enumeration we refer to the documentation provided with the package.

10.2.1 Constructing Funcon Terms

In the Haskell Funcon Framework, funcon terms are of type *Funcons*, of which values are constructed by applying so-called 'smart constructors'. Smart constructors are regular functions yielding datatype values. They are typically used to make datatypes abstract, hiding details of the chosen representation. The modules `Funcons.EDSL`, `Funcons.Core`, and `Funcons.MetaProgramming` export smart constructors for building funcon terms without revealing the details of the *Funcons* datatype. There is a smart constructor for each funcon in the library of funcons provided with CBS, generated by the CBS compiler. These smart constructors produce a funcon term given a list of funcon terms. As discussed in §8.1.5, strict, non-variadic funcons may be applicable to a different numbers of arguments then their arity suggests, as these arguments may rewrite to a sequence of the right length. We choose to treat all funcons equally in this regard, and give them all the general but uninformative type $[Funcons] \to Funcons$. As examples, consider the following smart constructors, generated for the funcons **sequential** and **if-true-else**:

sequential_ :: $[Funcons] \to Funcons$
if_true_else_ :: $[Funcons] \to Funcons$

The names of the smart constructors are identical to the funcons, except that hyphens are replaced by underscores, and they they have a trailing underscore to avoid overlap with other functions.

Built-in types, values and value operations are implemented in the `funcons-values` package. Smart constructors for built-in value operations have been developed manually and are exported by `Funcons.Core` in the same way as generated smart constructors. Similarly, each built-in type has a smart constructor, for example:

```
values_ :: Funcons                  -- builds terms of type types
lists_  :: [Funcons] → Funcons      -- builds terms of type types
```

Nullary built-in smart constructors, such as **values_** above, do not receive arguments

The `funcons-values` package uses Haskell values to represent the different types of values of CBS. There are smart constructors to build funcon terms from Haskell values, for example:

```
int_          :: Int                  → Funcons   -- of type integers
bool_         :: Boolean              → Funcons   -- of type booleans
string_       :: String               → Funcons   -- of type strings
env_fromlist_ :: [(String, Funcons)] → Funcons   -- of type maps(strings, values)
```

The function *env_fromlist_* builds **environments** given association lists in which strings (converted to **strings**/**identifiers**) are associated with funcon terms (which will be evaluated). This function is useful to construct an initial environment under which programs are executed, or to implement standard libraries, as shown by the Mini case study.

10.2.2 Executing Funcon Terms

The `Funcons.Tools` module exports functions for executing funcon terms, building funcon term interpreters, and composing funcon modules (see Section 9.1). The function *run* :: [*String*] → *Maybe Funcons* → *IO* () is given a list of command line arguments and an optional[1] funcon term for execution. The output is largely dependent on the command line arguments. The documentation of `funcons-tools` explains which command line options are available. Function *runWithExtensions* is a variation of *run* which receives additional funcon module components as arguments: funcon implementations, entity declarations, and type declarations. This is useful to extend the default funcon term interpreter, which only has implementations of the funcons provided with CBS, with language specific funcons, entities, and types.

The package `funcons-tools` builds an executable *runfct*, which accepts a configuration file (perhaps containing unit tests, see Section 9.4) or a file containing a funcon term as one of its command line arguments. As examples, consider the command line interactions on the next page.

[1] Optional, because a funcon term may also be provided as one of the command line arguments.

```
tmp/ ./runfct --funcon-term "initialise-giving give(3,sequential(print(integer-add
(1,given)),throw(given)))"
Result:
no-given(give(3,stuck))
Control Entity: abrupted
thrown(3)
Output Entity: standard-out
4

tmp/ cat example.fct
initialise-giving
  give(3,sequential(print(integer-add(1,given)),throw(given)))
tmp/ ./runfct example.fct
Result:
no-given(give(3,stuck))
Control Entity: abrupted
thrown(3)
Output Entity: standard-out
4

tmp/ cat example_test.config
general {
  funcon-term: initialise-giving
    give(3,sequential(print(integer-add(1,given)),throw(given)));
}
tests {
  result-term:  null-value;
  standard-out: [4];
}
tmp/ ./runfct example_test.config
expected result-term: null-value
actual   result-term: no-given(give(3,stuck))
unexpected abrupted: thrown(3)
```

Chapter 11

Case Study - Mini

Mini is a basic procedural programming language with arrays, exceptions, recursive procedures, and a check for uninitialised variables. The following program computes the first n Fibonacci numbers with dynamic programming and highlights most of the features of Mini:

```
1  procedure main (var n) {
2      var memo;
3      memo = empty_array(n+1);
4      memo[0] = 0;
5      memo[1] = 1;
6      memo[2] = 1;
7      procedure fib (var x)
8      begin
9          if (x <= 0) then return 0;
10         if (not (null memo[x]))
11           then return memo[x];
12           else
13             begin # compute the value and memoise it
14                 memo[x] = fib (x-1) + fib (x-2);
15                 return memo[x];
16             end
17     end
18     var i;
19     for i = 1; i <= n; i = i + 1;
20     begin
21         print (fib(i));
22     end
23 }
24 main(8);
```

Mini has several program constructs, categorised into expressions, commands, and declarations. In the next sections we explain each construct informally and define the syntax and semantics of each construct formally with the tools discussed in Chapter 10. We begin by listing all constructs, giving the symbol expressions that capture the syntax of expressions, commands, and declarations.

```
syn_expr = "expressions"   ⟨:=  syn_ops          -- prefix and infix operators
                           ⟨|||⟩ syn_parens      -- grouped expressions
                           ⟨|||⟩ syn_literals    -- literal values
                           ⟨|||⟩ syn_ident       -- identifiers
                           ⟨|||⟩ syn_proc        -- procedure invocation
                           ⟨|||⟩ syn_null        -- testing variable initialisation
                           ⟨|||⟩ syn_array       -- array construction
                           ⟨|||⟩ syn_array_idx   -- array indexing
syn_command = "commands"  ⟨:=⟩ syn_cmdexpr     -- expressions as commands
                           ⟨|||⟩ syn_print       -- printing output
                           ⟨|||⟩ syn_assign      -- variable assignment
                           ⟨|||⟩ syn_block_locals -- code blocks
                           ⟨|||⟩ syn_ite         -- if-then-else
                           ⟨|||⟩ syn_while       -- while-loops
                           ⟨|||⟩ syn_for         -- for-loops
                           ⟨|||⟩ syn_throw       -- throwing exceptions
                           ⟨|||⟩ syn_trycatch    -- catching exceptions
                           ⟨|||⟩ syn_return      -- returning values
syn_decl = "declarations" ⟨:=⟩ syn_vardecl     -- variable declarations
                           ⟨|||⟩ syn_procdecl    -- procedure declarations
```

11.1 Basic Expressions

Mini expressions evaluate to integers, booleans and strings. Integers are written as a non-empty sequence of digits. The sequence of characters `-123` is interpreted as the application of the prefix operator − to the integer `123`. Boolean values are written as either **true** or **false**. Strings are any sequence of characters in between double quotes. For a double quote to appear in a string, it has to be escaped using '\'.

```
syn_literals = "literals"
  ⟨:=⟩ int_          ⟨$$⟩ int_lit       -- non-empty sequence of digits
  ⟨|||⟩ bool_ True   ⟨$$ keyword "true"
  ⟨|||⟩ bool_ False  ⟨$$ keyword "false"
  ⟨|||⟩ string_      ⟨$$⟩ string_lit    -- characters within quotes
```

```
operator_semantics =
  [("built-in +",      infix_op [integer_add_ [given1, given2]])
  ,("built-in -",      infix_op [integer_subtract_ [given1, given2]])
  ,("built-in minus", prefix_op [integer_subtract_ [int_ 0, given_]])
  ,("built-in *",      infix_op [integer_multiply_ [given1, given2]])
  ,("built-in /",      infix_op [
             if_true_else_ [is_equal_ [int_ 0, given2]
               , throw_ [string_ "division-by-zero"]
               , integer_divide_ [given1, given2]]]
   )
  ,("built-in not", prefix_op [not_ [given_]])
  ,("built-in !",   prefix_op [not_ [given_]])
  ,("built-in ==",   infix_op [is_equal_ [given1, given2]])
  ,("built-in !=",   infix_op [not_ [is_equal_ [given1, given2]]])
  ,("built-in <",    infix_op [is_less_ [given1, given2]])
  ,("built-in <=",   infix_op [is_less_or_equal_ [given1, given2]])
  ,("built-in >",    infix_op [is_greater_ [given1, given2]])
  ,("built-in >=",   infix_op [is_greater_or_equal_ [given1, given2]])
  ,("built-in ^",    infix_op [string_append_ [given1, given2]])
  ]
infix_op body  = curry_ [prefix_op body]
prefix_op body = function_ [abstraction_ body]
given1         = first_ [tuple_elements_ [given_]]
given2         = second_ [tuple_elements_ [given_]]
```

Figure 11.1: Mini's built-in procedures and their implementations as **functions**.

Operators Infix operators have two operands evaluated in left to right order. Prefix operators have a single operand.

```
sem_infix e1 op e2 = apply_ [apply_ [bound_built_in op, e1], e2]
sem_prefix op e1   = apply_ [bound_built_in op, e1]
bound_built_in op  = bound_ [string_append_ [string_ "built-in ", string_ op]]
```

Operators are built-in procedures: at runtime a binding is active that binds a unique identifier (e.g. `"built-in +"` for +) to the body of a procedure. The procedure receives the values obtained by evaluating the operands and returns a result value. The precise bindings that are active for each operator are given in Figure 11.1.

An infix operator is written between two expressions and may be left-associative, right-associative or both. A prefix operator is written before an expression.

Operators have a relative precedence, determining which operator applications are evalu-

ated first. The fixity, precedence, and associativity of each operator are listed in the operator table given Figure 11.2 (operator tables are discussed in 10.1.3).

```
operator_syntax =
  [(0, [("not", Prefix sem_prefix)])
  ,(1, [("==", Infix  sem_infix LAssoc),("!=", Infix  sem_infix LAssoc)])
  ,(2, [("<",  Infix  sem_infix LAssoc),("<=", Infix  sem_infix LAssoc)
       ,(">",  Infix  sem_infix LAssoc),(">=", Infix  sem_infix LAssoc)])
  ,(3, [("+",  Infix  sem_infix LAssoc),("-",  Infix  sem_infix LAssoc)
       ,("^",  Infix  sem_infix LAssoc)])
  ,(5, [("*",  Infix  sem_infix LAssoc),("/",  Infix  sem_infix LAssoc)])
  ,(9, [("!",  Prefix sem_prefix)      ,("-",  Prefix sem_minus)])
  ]
syn_ops = fromOpTable "operators" (opTableFromList operator_syntax) syn_expr_lit
sem_minus _ e1 = sem_prefix "minus" e1
```

Figure 11.2: The precedence, fixity and associativity of Mini's operators.

Parentheses are used to group expressions.

```
syn_parens = parens syn_expr
```

11.2 Variables

Identifiers *bind* variables or procedures. Variables have values *assigned* to them. The same value can be assigned to multiple variables and multiple identifiers can bind the same variable. Bindings are temporary: they become active at a declaration site, and expire at the end of a block of code. Assignments do not expire (but a garbage collector may flush variables which are no longer bound). Bindings can be overwritten by binding the same identifier to a different variable or procedure. Assignments can be overwritten by assigning a different value to the same variable. The details of these concepts are discussed in this section. An identifier is a non-empty sequence of alphabetical characters and underscores.

Variable declarations Variable declarations are executed to activate a binding from an identifier to a variable. A variable declaration is written by the keyword **var** followed by an identifier and a semicolon.

```
syn_vardecl = "var-decl" ⟨:=⟩
                  sem_vardecl ⟨$$ keyword "var" ⟨**⟩ syn_id ⟨** keychar ';'
syn_id = string_ ⟨$$⟩ id_lit   -- alphabetical characters and underscores
```

Executing a variable declaration involves creating a new, uninitialised variable as a side-effect. The main result of executing a declaration is the activation of a binding: the identifier is bound to the newly created variable.

```
sem_vardecl i = bind_ [i, allocate_variable_ [values_]]
```

Expressions A Mini expression that evaluates to a variable is referred to as an *l-value* expression. There are two forms of l-value expressions. The first comprises a single identifier:

```
syn_ident   = sem_var ⟨$$⟩ syn_id
sem_var id = bound_ [id]
```

The second type of l-value expressions is introduced in Section 11.6.

If an l-value expression e occurs in a context where a value is expected, the variable to which e evaluates is dereferenced.

```
syn_expr_lit  = ensure_lit ⟨$$⟩ syn_expr
ensure_lit e1 = else_ [current_value_ [e1]
                      , sem_throw "uninitialised-reference-component"]
```

A Mini expression that evaluates to a (non-variable) value is referred to as an *r-value* expression. For example, the expression x+y is an r-value expression, and the subexpressions x and y are r-value expressions as well.

Mini programs in which an identifier occurs in a place where it is not bound are not valid. If an expression is evaluated in which an identifier is bound to an uninitialised variable, then the `"uninitialised-reference-component"` exception (see Section 11.5) is raised. Programs in which r-value expressions occur in a context where an l-value expression is expected are not valid.

A programmer can check whether a variable is initialised by writing **null** e, where e is an l-value expression. This Boolean expression evaluates to **true** if e evaluates to an uninitialised variable and **false** otherwise.

```
syn_null     = sem_null ⟨$$ keyword "null" ⟨**⟩ syn_expr
sem_null e1 = else_ [sequential_ [effect_ [assigned_ [e1]], bool_ False], bool_ True]
```

For example, the following program prints **true** twice:

```
1 var x;
2 print (null x); # prints true
3 x = 1;
4 print (not (null x)); # prints true
```

11.3 Basic Commands

Commands are executed for their effects: printing values and mutating the assignment of simple or composite variables (see Section 11.6).

The print command is written as the keyword **print** followed by an r-value expression and a semicolon.

$$\mathit{syn\_print} = \mathit{sem\_print}\ \langle\$\$\ \mathit{keyword}\ \texttt{"print"}\ \langle{**}\rangle\ \mathit{syn\_expr\_lit}\ \langle{**}\ \mathit{keychar}\ \texttt{';'}$$

The expression is evaluated and its values is translated into a string. The string is extended with the newline character and printed to the standard output.

$$\begin{aligned}\mathit{sem\_print}\ e_1 = \mathsf{print\_}\ [\mathsf{string\_append\_}\ [\mathsf{mini\_show\_}\ [e_1] \quad &\text{-- defined in Section 11.8}\\ , \mathit{string\_}\ \texttt{"\textbackslash n"}]]\end{aligned}$$

An assignment is an l-value expression followed by the =-operator and an r-value expression followed by a semicolon.

$$\begin{aligned}\mathit{sem\_assign}\ e_1\ e_2 &= \mathsf{assign\_}\ [e_1, e_2]\\ \mathit{syn\_assign} &= \texttt{"assignment"}\ \langle{:=}\rangle\ \mathit{sem\_assign}\ \langle\$\$\rangle\\ &\quad\ \mathit{syn\_expr}\ \langle{**}\ \mathit{keychar}\ \texttt{'='}\ \langle{**}\rangle\ \mathit{syn\_expr\_lit}\ \langle{**}\ \mathit{keychar}\ \texttt{';'}\end{aligned}$$

For example, x = 3+y; is a valid assignment, but x<0 = 3+y; is not, because x<0 is not an l-value expression.

Mini expressions may have side-effects when they involve procedure invocation (see Section 11.4). An expression can be evaluated for its effects only, by executing the expression as a command. An r-value expression followed by a semicolon is a command.

$$\begin{aligned}\mathit{sem\_cmdexpr}\ e_1 &= \mathsf{effect\_}\ [e_1]\\ \mathit{syn\_cmdexpr} &= \mathit{sem\_cmdexpr}\ \langle\$\$\rangle\ \mathit{syn\_expr\_lit}\ \langle{**}\ \mathit{keychar}\ \texttt{';'}\end{aligned}$$

Blocks of code A Mini program (see Section 11.7) is a sequence of globals, where a global is a declaration or a command. A sequence of globals is executed in order, so that the side-effects of each global can be witnessed by subsequent globals. On top of that, a declaration makes a new binding available to subsequent globals.

```
syn_globals = "globals" ⟨:=⟩ foldr_multiple syn_global null_
syn_global :: SymbExpr Token (Funcons → Funcons)
syn_global  = "global"  ⟨:=⟩ activate_in ⟨$$⟩ syn_decl
                        ⟨|||⟩ and_then   ⟨$$⟩ syn_command
activate_in env fct = scope_ [env, fct]
and_then cmnd fct = sequential_ [cmnd, fct]
```

A code block is also a sequence of declarations and commands, delimited by braces or by the keywords **begin** and **end**.

```
syn_locals = syn_globals
syn_local  = syn_global
syn_block_locals = "block-locals" ⟨:=⟩ keyword "begin" **⟩ syn_locals ⟨** keyword "end"
                                  ⟨|||⟩ braces syn_locals
```

A code block may occur in place of a command, as in the following program:

```
1 var x;
2 begin
3   x = 1;
4   var y;
5   print x;
6 end
7 print (not (null x));
```

The lifetime of a variable ends at the end of the code block in which it is declared. Therefore y is not in scope after line 6 in the program above. Identifier x is in scope however, and line 7 prints `true`, because assignments persist.

Code blocks can also appear as components of a control-flow command (Section 11.5). In this case, a single command (without delimiters) can be written instead of a code block.

```
syn_branch = "component-block" ⟨:= syn_block_locals
                               ⟨|||⟩ (flip ($) null_) ⟨$$⟩ syn_local
```

As an example, consider the following program with an **else**-branch formed out of a single command.

```
1 var x;
2 if (null x) then begin
3     x = 1;
4     print "x is set";
5   end
6   else
7     print "x was set";
```

11.4 Procedures

A user-defined procedure is *declared* once and can subsequently be *invoked* zero or more times. A procedure is declared by writing the keyword **procedure** followed by the procedure's name — an identifier — a comma-separated (possibly empty) sequence of variable declarations within parentheses, and the procedure's body — a code block. The variables mentioned within a procedure declaration are the procedure's formal parameters.

```
syn_procdecl = sem_procdecl ⟨$$
  keyword "procedure" ⟨**⟩ syn_id ⟨**⟩ parens syn_formals ⟨**⟩ syn_branch
syn_formals :: SymbExpr Token [Funcons]
syn_formals = "formals" ⟨:=⟩ multipleSepBy (keyword "var" **⟩ syn_id) (keychar ',')
```

A procedure declaration binds the name of the procedure to its body. When invoked, the formal parameters (if any) are bound to the arguments given to the procedure. As shown later, the arguments are fresh, initialised variables.

```
sem_procdecl id params body = bind_recursively_ [id, abs]
  where abs | null params = closure_ [body]
            | otherwise   = foldr combine body params
        combine param code =
          function_ [closure_ [scope_ [bind_ [param, given_], code]]]
```

A procedure may be invoked recursively, i.e. the body of a procedure may contain an invocation of the procedure itself. A procedure invocation is an expression written as the procedure's name followed by the actual parameters, a comma-separated sequence of r-value expressions within parentheses.

```
syn_proc = sem_invoke ⟨$$⟩ syn_id ⟨**⟩ parens syn_actuals
syn_actuals :: SymbExpr Token [Funcons]
syn_actuals = "actuals" ⟨:=⟩ multipleSepBy syn_expr_lit (keychar ',')
```

A procedure invocation evaluates to the value returned by the procedure. The evaluation involves finding the procedure body bound to the identifier, and evaluating the actual parameters (if any). The values that result are assigned to fresh variables, forming the arguments that are given to the procedure body.

```
sem_invoke id args = handle_return_ [invocation]
  where invocation | null args  = enact_ [bound_ [id]]
                   | otherwise = foldl app (bound_ [id]) args
        app abs arg = apply_ [abs, allocate_initialised_variable_ [values_, arg]]
```

A return command within a procedure body terminates procedure invocations, potentially before all declarations and commands in the body have been executed. A return command is written as the keyword **return** followed by an r-value expression and a semicolon.

```
sem_return e1 = return_ [e1]
syn_return    = sem_return ⟨$$ keyword "return" ⟨**⟩ syn_expr_lit ⟨** keychar ';'
```

Programs in which a procedure with n formal parameters is invoked with $m \neq n$ actual parameters are invalid. A procedure without a return command is fully executed, and returns no value as a result. A procedure which is not guaranteed to return a value may only be invoked directly as a command, not as part of another expression. If a procedure invocation raises an exception, the invocation is terminated and the exception is propagated. The following example program produces no output:

```
1  var a;
2  a = 1;
3  procedure noreturn (var a, var b, var c) {
4    if (null a) then print "ERROR";
5    if (null b) then print "ERROR";
6    if (null c) then print "ERROR";
7  }
8  noreturn(1,2,3);
9
10 procedure max (var a, var b, var c) {
11   if (a >= b) then if (b >= c) then return a;
12   if (b >= c) then return b;
13   return c;
14 }
15 if (max(1,2,3) < 3) then print "ERROR";
```

11.5 Control Flow

Mini has several control-flow constructs. The basic constructs determine which code to execute based on Boolean conditions.

Basic Flow The **if**−**then**−**else**-command is the **if** keyword followed by an r-value expression — the condition — and two code blocks, preceded by **then** and **else** respectively, of which the latter is optional.

```
sem_ite cond t e = if_true_else_ [cond, t, e]
syn_ite          = sem_ite ⟨$$ keyword "if" ⟨**⟩ syn_expr_lit
                           ⟨** keyword "then" ⟨**⟩ syn_branch
                           ⟨**⟩⟩⟩ optionalWithDef (keyword "else" **⟩ syn_branch) null_
```

To execute an **if**−**then**−**else**-command, the condition is evaluated and, based on its value, one of the code blocks is executed. An occurrence of **if**−**then**−**else** without an **else**-block is syntactic sugar for **if**−**then**−**else** with an empty **else**-block.

A **while**-loop is a command written as the **while** keyword followed by a condition and a code block. A **while** command is executed by evaluating its condition and executing its body until the condition no longer holds. The condition is checked before every iteration; if the condition is **false**, the body is not executed at all.

```
sem_while cond body = while_ [cond, body]
syn_while           = sem_while ⟨$$ keyword "while" ⟨**⟩ syn_expr_lit ⟨**⟩ syn_branch
```

A **for**-loop is a command written as the **for** keyword followed by three commands and a code block. The commands must be an assignment — the initialiser — a Boolean r-value expression — the condition — and another assignment — the updater. A **for**-loop is syntactic sugar for a **while**-command in the conventional way: executing a **for**-loop is executing the initialiser, followed by a **while**-loop with the condition of the **for**-loop and a body which is the **for**-loop's body extended with the updater.

```
syn_for = sem_for ⟨$$ keyword "for" ⟨**⟩
  syn_assign ⟨**⟩ syn_expr ⟨** keychar ';' ⟨**⟩ syn_assign ⟨**⟩ syn_branch
sem_for init cond upd body = and_then init (sem_while cond (and_then body upd))
```

The following programs are equivalent, printing the first 10 integers.

```
1 var i;
2 i = 0;
```

```
3 while i < 10 begin
4   print (i);
5   i = i+1;
6 end
```

```
1 var i;
2 for i = 0; i < 10; i = i + 1; print(i);
```

Exceptional Flow Mini has a basic mechanism for throwing and catching exceptions. An exception is a string, thrown by the **throw**-command, which is written as the keyword **throw** followed by an r-value expression and a semicolon.

```
sem_throw str = throw_ [string_ str]
syn_throw     = sem_throw ⟨$$ keyword "throw" ⟨**⟩ string_lit ⟨** keychar ';'
```

Programs in which the expression of a **throw**-command does not evaluate to a string are invalid. Throwing an exception terminates the execution of the surrounding code block, or of the program, if there is no surrounding code block. An exception is propagated, eventually terminating the execution of the whole program, unless the exception occurs in the **try**-block of a **try-catch**-command, in which case it may be caught.

The **try-catch**-command is a series of code blocks, the first of which is the **try**-block, the others are *handlers*. Syntactically, the **try**-block is preceded by the **try** keyword, and the handlers are preceded by the **catch** keyword, followed by a string literal.

```
syn_trycatch = sem_try ⟨$$ keyword "try" ⟨**⟩ syn_branch ⟨**⟩ syn_handlers
sem_try tryblock handlers = handle_thrown_ [tryblock, handlers]
syn_handlers = "handlers" ⟨:=⟩ foldr_multiple syn_handler (throw_ [given_])
syn_handler :: SymbExpr Token (Funcons → Funcons)
syn_handler = "single-handler"
  ⟨:=⟩ sem_handler ⟨$$ keyword "catch" ⟨**⟩ string_lit ⟨**⟩ syn_branch
sem_handler exc body handlers =
  if_true_else_ [is_equal_ [string_ exc, given_], body, handlers]
```

Executing a **try-catch**-command begins with executing its **try**-block. The command has finished if the **try**-block is executed without throwing an exception. If an exception x is thrown, then the handlers are considered, in order, until the first handler that is applicable to x is found. A handler is applicable to x if the string literal associated with the handler equals x. When applicable, the handler is executed. If executing a handler throws an exception, this exception is propagated. If no handler is applicable to x, x is propagated.

Not only user-thrown exceptions can be caught. For example, the following program has handlers for `"uninitialised-reference-component"` and `"division-by-zero"` exceptions:

```
1  var x;
2  try begin
3    print x;
4    print "ERROR";
5    print (0 / 0);
6  end
7  catch "uninitialised-reference-component" begin
8    print "x not initialised";
9  end
10 catch "division-by-zero"
11   print "ERROR";
```

11.6 Arrays

Besides integers, Booleans, and strings, Mini also has arrays. An array is an example of a *composite variable*: a collection of variables with a particular structure. An array has a fixed length n and there is a variable associated with each index 0 to $n - 1$. An array is an ordinary value, and can thus be assigned to a variable, returned by a procedure, etc. To create arrays, Mini has array-notation: a comma-separated sequence of r-value expressions surrounded by brackets.

```
syn_array = "array" ⟨:=⟩
  sem_array ⟨$$⟩ brackets (multipleSepBy syn_expr_lit (keychar ','))
sem_array elems = vector_ [left_to_right_ (map alloc elems)]
  where alloc e = allocate_initialised_variable_ [values_, e]
```

The expressions are evaluated in left-to-right order, and each resulting value is assigned to a fresh variable. For example, the expression `[1,2,3,4,5]` creates an array consisting of five variables with indices 0 to 4, initialised with the numbers 1 to 5.

So far we have seen one example of an l-value expression: an expression containing just an identifier bound to a variable. Array indexing is an example of another l-value expression. An array indexing expression is written as an identifier and an r-value expression within brackets. In valid programs, the identifier is bound to a variable with an array assigned to it, and the expression evaluates to an integer.

```
syn_array_idx = "array-index" ⟨:=⟩ sem_array_idx ⟨$$⟩ syn_id ⟨**⟩ brackets syn_expr_lit
```

Evaluating an l-value expression $a[e]$ involves retrieving the array assigned to the variable to which a is bound, and evaluating e to i. If i is not an index of the array, an `"array-index-out-of-bounds"` exception is thrown. If it is, the variable with index i of the array is the result of evaluating the l-value expression.

```
sem_array_idx id e1 = give_ [e1, if_true_else_ [range, vector_index given_ vec
                                                , sem_throw "array-index-out-of-bounds"]]
  where vec = ensure_lit (sem_var id)
    range     = and_ [is_greater_or_equal_ [given_, int_ 0]
                      , is_less_ [given_, vector_length vec]]
vector_length v  = length_ [vector_elements_ [v]]
vector_index i v = index_ [integer_add_ [int_ 1, i], vector_elements_ [v]]
```

Because array indexing is an l-value expression, it can occur in places where a value is expected (see Section 11.2), as well as on the left-hand side of an assignment. For example, the following program creates an array of length one and updates and accesses the variable it consists of:

```
1 var arr;
2 arr = [1];
3 arr[0] = 2;
4 print (arr[0]);
```

Operations on arrays In Section 11.1, several built-in procedures were introduced. The following additional built-in procedures are available for working with arrays:

```
array_built_ins =
  [("array_length", prefix_op [vector_length (assigned_ [given_])])
  ,("empty_array", prefix_op [vector_
      [interleave_repeat_ [allocate_variable_ [values_], int_ 1, assigned_ [given_]]]])]
```

The procedure `array_length` is given an argument assigned to an array and returns the length of that array. The procedure `empty_array` receives as argument a variable assigned to n and returns an array of length n, consisting of n uninitialised variables. The following programs shows how these procedures are used:

```
1 var y;
2 y = empty_array(2);
3 print (null y[0]); # prints true
```

```
print (null y[1]); # prints true
y[0] = 1;
y[1] = 2;
print y[1]; # prints 2
print y[0]; # prints 1
print array_length(y); # prints 2
y[2] = 3; # throws "array-index-out-of-bounds" exception
```

11.7 Programs

A Mini program is a sequence of declarations and commands executed in order.

```
syn_program = "program" ⟨:=⟩ sem_program ⟨$$⟩ syn_globals
sem_program body =
  initialise_binding_ [initialise_storing_ [
    handle_thrown_ [scope_ [all_built_in_identifiers, body], handler]]]
  where handler = print_ [string_append_
                            [string_ "Uncaught Exception: "
                            , given_, string_ "\n"]]
all_built_in_identifiers = env_fromlist_ $ array_built_ins ++ operator_semantics
```

A program is executed as a code block (see Section 11.3). However, if a program terminates because an exception is thrown, the exception is reported. For example, the message `Uncaught Exception: division-by-zero` is the output of the following program:

```
var x;
var y;
y = 3;
x = 1;
print (y / ((x * 42) - 7*6));
```

11.8 Interpretation

This section discusses how the Haskell code segments from the previous sections form an interpreter for Mini. The interpreter is available as the `mini-reuse` package in the supplementary material. The function $lexer :: String \to [Token]$ is omitted. The parser is defined as follows, applying *parse* from the `gll` package to *syn_program*:

```
parser :: [Token] → [Funcons]
parser = parse syn_program
```

Note that running the parser may result in several funcon terms, one for each possible interpretation of the input string. Although several disambiguation strategies have been identified, e.g. longest-match to solve the 'dangling else problem' (see Section 11.5), there is no guarantee that there is at most one interpretation for every program. The *main* function defined below combines the lexer and parser to obtain zero or more funcon terms, and applies the function *runWithExtensions* imported from `Funcons.Tools` to execute them.

```
main = do
  args ← getArgs
  case args of
    []      → putStrLn "Please provide me with an input file"
    f : opts → go f opts
where
  go :: FilePath → [String] → IO ()
  go f opts = do
    str ← readFile f
    let tokens    = lexer str
        fcts      = parser tokens
        ambiguous = length fcts > 1
    forM_ (zip [1..] fcts) $ λ(i, fct) → do
      when ambiguous (putStrLn ("=== Interpretation " ++ show i ++ "\n"))
      runWithExtensions Mini.funcons Mini.entities Mini.types opts (Just fct)
```

The first command line argument is expected to be a Mini program. The other command line arguments are passed on to *runWithExtensions* from the Haskell Funcon Framework. The function *runWithExtensions* runs the funcon term interpreter on a given funcon term, extending the interpreter with the implementations of additional, language-specific funcons. The only Mini-specific funcon **mini-show** is defined below, and its implementation is exported by the module `Funcons.Mini.Mini`, imported under the qualified name *Mini*.

Funcon **mini-show**(V : **values**)

Rule **mini-show**(**true**) ⇝ `"true"`

Rule **mini-show**(**false**) ⇝ `"false"`

Rule **mini-show**(I : **integers**) ⇝ **to-string**(I)

Rule **mini-show**(S : **strings**) ⇝ S

Rule **mini-show**(**vectors**(V^*)) ⇝

string-append(`"["`, **intersperse**(`", "`,

interleave-map(**mini-show**(**assigned**(**given**)), V^*)), `"]"`)

Chapter 12

Case Study - Caml Light

This chapter follows roughly the structure[1] of the Caml Light reference manual [Leroy, 1997]. We describe the syntax of Caml Light with the BNF combinators, following the definitions of the reference manual closely. As for Mini, we associate semantic functions with each syntactic construct, and the semantic functions yield funcon terms which can be executed within the Haskell Funcon Framework. Parts of the funcon translation are derived from the CSF specification[2] of Caml Light presented in [Churchill et al., 2015], and from the superseding CBS specification subsequently developed by Neil Sculthorpe. Version 0.74 of Caml Light is described, together with some of the extensions suggested in the reference manual. The package `caml-light-reuse` is constructed from the sources of this chapter and is part of the supplementary material. The funcon translation depends on a small number of Caml Light specific funcons defined in CBS.

12.1 Lexical Conventions

The precise specification of lexical items and the development of lexers are not within the scope of this thesis, and we omit these here. The function $lexer : String \to [\,Token\,]$, used by the interpreter of Section 12.9, is exported by the module `Lexer`. Compared to [Leroy, 1997],

[1] For clarity, Sections 12.3 and 12.2 are swapped. The sections on global definitions and module implementations have merged. The sections on module interfaces and directives have been omitted.

[2] CSF preceded CBS.

we have simplified the lexical conventions regarding identifiers and literals in a number of ways. Most importantly, we distinguish identifiers based on whether they start with a lowercase or uppercase alphabetical character[3]. Identifiers starting with an uppercase letter are reserved for the constructors of variant values (see Section 12.2). As described by Leroy, an identifier can appear both as a variable and as a constructor name. The latter interpretation is to be taken if the identifier is introduced as a constructor earlier in the program. For example, consider the following program:

```
1 type winddir = north | east | south | west ;;
2
3 (fun x -> 0) ;;       (* x an identifier *)
4 (fun north -> 0) ;;   (* north a constant constructor *)
```

According to [Leroy, 1997], identifier `north` on line 4 should be interpreted as a constructor. However, if line 1 is omitted, it should be interpreted as a variable instead. We determine that constructor names must begin with uppercase letters. The program above is thus not valid according to our syntax description. Instead the program is written as:

```
1 type winddir = North | East | South | West ;;
2
3 (fun x -> 0) ;;       (* x an identifier *)
4 (fun North -> 0) ;;   (* north a constant constructor*)
```

`North` on line 4 is interpreted as a constructor, also if line 1 is removed, simply because it starts with an uppercase later (see Section 12.6 on patterns).

Other differences with [Leroy, 1997] are more subtle. For example, numeric literals (float literals and integer literals) cannot start with a negative (or positive) sign; numeric literals always denote numbers $\geqslant 0$. For example, -1 is interpreted as the application of the prefix operator $-$ to the integer literal 1. String literals are also simplified: within a string, a character sequence of a backslash and three digits is interpreted as just that, rather than the code of an ascii-character.

[3]Tokens of the first kind are produced by *id_lit* and of the latter kind by *alt_id_lit*.

12.2 Values

The base types of Caml Light are integers, floating-point numbers, characters and character strings. We place no restrictions on the size of integers and use the Haskell values of types *Int*, corresponding to the representation of integers by the `gll` and `funcons-values` packages. Similarly, we use values of type *Double* to represent floats and values of type *Char* to represent characters. Strings are lists of characters.

The composite types of Caml Light are tuples, lists, records, arrays, variants, and functions. The components of arrays, variants, records and strings are mutable. There are no restrictions on the sizes of tuples, strings, records, and arrays. A variant value is a pair of a constructor name and an optional argument. If a constructor has an arity greater than one, its argument is a tuple holding multiple values. A constructor with arity zero is a constant.

12.3 Global Names

Global names are identifiers possibly qualified with a module name (also an identifier):

```
syn_global :: SymbExpr Token Funcons
syn_global = "global-name" ⟨:=⟩ string_ ⟨$$ optional (id_lit ⟨** keyword "__") ⟨**⟩ id_lit
syn_cap_global :: SymbExpr Token Funcons
syn_cap_global =
  "cap-global-name" ⟨:=⟩ string_ ⟨$$ optional (id_lit ⟨** keyword "__") ⟨**⟩ alt_id_lit
```

Global names with capitalised identifiers are reserved for constructor names. Qualified identifiers are only supported in syntax — any qualifiers are ignored semantically. We do not consider opening and closing modules.

A variable is a global name or is a name reserved for an operator.

```
syn_var :: SymbExpr Token Funcons
syn_var = "variable" ⟨:=⟩ syn_global
                     ⟨||⟩ sem_op_var ⟨$$ keyword "prefix" ⟨**⟩ syn_op
where
  syn_op :: SymbExpr Token String
  syn_op = chooses    -- chooses s [a1,...,an] ≡ s ⟨:=⟩ a1 ⟨||⟩ ... ⟨||⟩ an
             "operator-name" (map keyword operator_names)
sem_op_var op = string_ ("prefix " ++ op)
```

For example, the variable `prefix +` refers to the value to which `prefix +` is currently bound. Initially, this is a function value, adding two integers when applied, according to

```
operator_names = ["+" ,"*" ,"/"  ,"mod","+." ,"*.","/." ,"-" ,"-."
                 ,"@" ,"^" ,"!"  ,":=" ,"="  ,"<>","==" ,"!=","<"
                 ,"<=",">" ,">=" ,"<." ,"<=.",">.",">=.","**","::"
                 ,"-" ,"-.","not","or" ,"&&" ,"&" ,"||" ,"or"]
```

Figure 12.1: Operators that can be redefined, originally defined in the core library.

the specification of the core library (see Chapter 13 of [Leroy, 1997]), in which it is defined. However, this binding may be overridden by the programmer. All the functions of the core library may be overridden this way. Its operators are listed in Figure 12.1.

A global name can also appear as the field identifier (label) of a record (records and fields are explained near the end of Section 12.7):

```
syn_label :: SymbExpr Token Funcons
syn_label = "label" ⟨:=⟩ syn_global
```

A capitalised global name appears as a constant constructor or non-constant constructor.

```
syn_cconstr :: SymbExpr Token Funcons
syn_cconstr = "cconstr" ⟨:=⟩ sem_cconstr_global ⟨$$⟩ syn_cap_global
                        ⟨||⟩ list_ []           ⟨$$ keychar '[' ⟨** keychar ']'
                        ⟨||⟩ vector_ []         ⟨$$ keyword "[|" ⟨** keyword "|]"
                        ⟨||⟩ null_              ⟨$$ keychar '(' ⟨** keychar ')'
                        ⟨||⟩ bool_ True         ⟨$$ keyword "true"
                        ⟨||⟩ bool_ False        ⟨$$ keyword "false"
  where sem_cconstr_global g = bound_ [g]
syn_ncconstr :: SymbExpr Token Funcons
syn_ncconstr = "ncconstr" ⟨:=⟩ syn_cap_global
```

The constant constructors `[]`, `[||]`, and `()`, for the empty list, empty array, and empty tuple respectively, are built-in. The Boolean values `true` and `false` are also constants. A non-constant (value) constructor is the name of a function that yields a variant (a value of an algebraic datatype).

Type expressions (Section 12.4) can be formed by applying a type constructor to zero or more type expressions. A type constructor name starts with a lowercase character.

```
syn_tycons :: SymbExpr Token Funcons
syn_tycons = "typeconstr" ⟨:=⟩ syn_global
```

12.4 Type Expressions

Type expressions occur as pattern and expression annotations, providing static information about the annotated pattern or expression. This information is helpful to programmers and can guide type-inferencing. Type expressions do not influence dynamic semantics and we only consider their syntax here.

```
syn_tyexpr :: SymbExpr Token ()
syn_tyexpr = "typexpr"
  ⟨::=⟩ () ⟨$$ syn_tyexpr ⟨** keyword "->" ⟨⟨⟨** syn_tyexpr
  ⟨||⟩ () ⟨$$ syn_tyexpr ⟨** keyword "*"  ⟨⟨⟨** syn_tyexpr
  ⟨||⟩ () ⟨$$ syn_tycons
  ⟨||⟩ () ⟨$$ syn_tyexpr ⟨** syn_tycons
  ⟨||⟩ () ⟨$$ parens (multipleSepBy1 syn_tyexpr (keyword ",")) ⟨** syn_tycons
  ⟨||⟩ () ⟨$$ keyword "'" ⟨** id_lit
  ⟨||⟩ parens syn_tyexpr
```

Compared to the applications of functions and constructors, type constructor applications are written in reversed order; the arguments precede the type constructor.

12.5 Constants

Constants are the literal values of the base types — integers, floats, characters and strings — and constant constructors (see Section 12.3):

```
syn_const :: SymbExpr Token Funcons
syn_const = "literal" ⟨:=⟩ int_        ⟨$$⟩ int_lit      -- integer literal to funcon term
                       ⟨||⟩ sem_string ⟨$$⟩ string_lit
                       ⟨||⟩ float_     ⟨$$⟩ float_lit    -- float literal to funcon term
                       ⟨||⟩ char_      ⟨$$⟩ syn_char     -- char literal to funcon term
                       ⟨||⟩ syn_cconstr

syn_char :: SymbExpr Token Char   -- replaces char_lit for different literal form
syn_char = "char" ⟨:=⟩ head ⟨$$⟩ token "CHAR"

sem_string cs = list_ (map mkMut cs)   -- a string is a list of variables storing characters
  where mkMut c = allocate_initialised_variable_ [characters_ [], char_ c]
```

The function *syn_char* is introduced to be used instead of *char_lit* because *char_lit* recognises character literals written between quotes (in the style of Haskell), whereas Caml Light character literals are written between backticks (*head* is used to extract the character from the token string). The translation of string literals (*sem_string*) shows that strings are sequences of mutable characters.

```
syn_pat :: SymbExpr Token Funcons
syn_pat = "pattern"
  ⟨::= syn_pat_as
  ⟨||⟩ syn_pat_alt
  ⟨||⟩ syn_pat_prod
  ⟨||⟩ syn_pat_cons
  ⟨||⟩ syn_pat_app
  ⟨||⟩ syn_pat_wildcard
  ⟨||⟩ syn_pat_id
  ⟨||⟩ parens (syn_pat ⟨** optional (keyword ":" ⟨** syn_tyexpr))
  ⟨||⟩ syn_pat_const
  ⟨||⟩ syn_pat_rec
  ⟨||⟩ syn_pat_list
  ⟨||⟩ syn_range
```

Figure 12.2: The syntax of Caml Light patterns.

12.6 Patterns

Values can be matched against patterns in order to perform case analysis and to bind identifiers to components of values of composite types. The complete syntax of patterns is given in Figure 12.2. The different forms of patterns are explained in what follows.

Matching a value against a pattern either fails or succeeds, potentially binding identifiers to components of the matched value. As we shall see in Section 12.7, a pattern match failure causes the `Match_failure` exception to be thrown.

Matching against an identifier always succeeds, binding the identifier to the matched value. This value could be the mutable component of a composite type.

```
syn_pat_id    = sem_pat_id ⟨$$⟩ id_lit
sem_pat_id i = pattern_bind_ [string_ i]
```

The wildcard pattern `_` matches any value without activating new bindings.

```
syn_pat_wildcard  = sem_pat_wildcard ⟨$$ keyword "_"
sem_pat_wildcard = pattern_any_
```

With the **as** keyword, a pattern can be annotated with an identifier. When a value v is matched against pattern p **as** i, v is matched against p. Any bindings activated by matching v against p are extended with i binding v.

```
syn_pat_as          = sem_pat_as ⟨$$⟩ syn_pat ⟨** keyword "as" ⟨**⟩ id_lit
sem_pat_as p nm = pattern_unite_ [p, pattern_bind_ [string_ nm]]
```

Matching a value v against a constant c succeeds if v is equal to c and fails otherwise.

```
syn_pat_const :: SymbExpr Token Funcons
syn_pat_const = "constant-pattern" ⟨:= sem_pat_string ⟨$$⟩ string_lit
                                   ⟨||⟩ match_constant ⟨$$⟩ syn_const
  -- structural_assigned_ is required to match the values held by mutable components
match_constant p = pattern_ [closure_ [match_ [structural_assigned_ [given_], p]]]
sem_pat_string s  = match_constant (string_ s)   -- no need to initialise variables
```

Multiple patterns can form 'products' or 'sums' when separated by commas or vertical bars respectively. When a value v is matched against a product pattern p with components $p_1 \ldots, p_n$, v must be an n-tuple with components $v_1 \ldots, v_n$ such that v_i matches p_i, for all $1 \leqslant i \leqslant n$. If matching any component fails, matching v against p fails. Otherwise, all the bindings resulting from matching the components unite to form the result of matching v against p.

```
syn_pat_prod = shortest_match (sem_pat_prod ⟨$$⟩ multipleSepBy2 syn_pat (keyword ","))
sem_pat_prod ps = tuple_ ps
```

When a value v is matched against a pattern p of the form $p_1 \mid p_2$, v is first matched against p_1. Only if this fails is v matched against p_2. Matching v against p either fails, is equivalent to matching v to p_2, if v does not match p_1, or is equivalent to matching v against p_1.

```
syn_pat_alt       = sem_pat_alt ⟨$$⟩ syn_pat ⟨** keyword "|" ⟨**⟩⟩ syn_pat
sem_pat_alt p1 p2 = pattern_else_ [p1, p2]
```

Caml Light supports two methods for decomposing lists by pattern matching. The first method is a pattern written with list-notation, e.g. $[pat_1; \ldots; pat_n]$.

```
syn_pat_list    = sem_pat_list ⟨$$⟩ brackets (multipleSepBy1 syn_pat (keyword ";"))
sem_pat_list ps = list_ ps
```

The second method is a pattern formed by applying list constructor :: , e.g. $pat_1 :: pat_2$.

```
syn_pat_cons = sem_pat_cons ⟨$$⟩ syn_pat ⟨** keyword "::" ⟨⟨**⟩ syn_pat
sem_pat_cons p1 p2 =
  pattern_ [closure_ [if_true_else_ [is_equal_ [given_, list_ []], fail_,
                                     collateral_ [match_ [head_ [given_], p1], match_ [tail_ [given_], p2]]]]]
```

When matching a list v with elements $v_1, \ldots, v_n$ against a pattern p of the form $p_1 :: p_2$, the head of v (element v_1) is matched against p_1 and the tail of v (the list containing $v_2, \ldots, v_n$) against p_2. As with tuples, if any component fails to match, matching v against

p fails. Otherwise, the bindings produced by matching components unite to form the result of matching v against p. If $n = 0$, i.e. v is the empty list, matching v against p fails.

A record pattern is of the form $\{label_1 = pat_1, \ldots, label_n = pat_n\}$.

```
syn_pat_rec = sem_pat_rec ⟨$$⟩ braces (multipleSepBy1 syn_entry (keyword ";"))
  where syn_entry :: SymbExpr Token (Funcons, Funcons)
        syn_entry = "pat-entry" ⟨:=⟩ (,) ⟨$$⟩ syn_label ⟨** keyword "=" ⟨**⟩ syn_pat
sem_pat_rec kvs = pattern_ [closure_ [match_loosely_ [given_, record_ [map_unite_ ents]]]]
  where ents = map (λ(l, p) → map_ [tuple_ [l, p]]) kvs
```

A record v with component values $v_1, \ldots, v_n$, labelled $l_1, \ldots, l_n$ respectively, matches a pattern p if p is a record pattern with component patterns $p_1, \ldots, p_m$, labelled $l'_1, \ldots, l'_m$ respectively, if $\{l'_1, \ldots, l'_m\} \subseteq \{l_1, \ldots, l_n\}$, and if for each $1 \leqslant 1 \leqslant m$ x_i matches p_i, where x_i is the value labelled with l'_i. As with tuples and lists, matching v against p fails if matching any component fails. If all components match, the resulting bindings unite to form the result of matching v against p.

A pattern can be formed by applying a non-constant value constructor to a pattern (which may be a tuple pattern). This enables programmers to match variant values of user-defined types.

```
syn_pat_app          = sem_pat_app ⟨$$⟩ syn_ncconstr ⟨**⟩ syn_pat
sem_pat_app nm p = variant_ [nm, p]
```

A variant value v matches variant pattern p if they are constructed by the same value constructor, and if their arguments match.

Finally, Caml Light has 'range patterns' of the form c_1 .. c_2, where c_1 and c_2 are characters constants. For example, the pattern `'0'..'9'` matches all digits. In general, a range pattern of the form c_1 .. c_n is a shorthand for the pattern $c_1 \mid \ldots \mid c_n$ containing the alternate patterns $c_1, \ldots, c_n$ such that $c_1, \ldots, c_n$ are all the characters that appear within c_1 and c_n (inclusive) in the ASCII character set:

```
syn_range = sem_pat_range ⟨$$⟩ syn_char ⟨** keyword ".." ⟨**⟩ syn_char
sem_pat_range c1 c2 = foldr combine fail_ [c1 .. c2]
  where combine c fp = sem_pat_alt (char_ c) fp
```

In a function definition (see Section 12.7), several patterns may occur one after the other, matching the potentially multiple arguments to which the function may be applied:

```
syn_pats :: SymbExpr Token [Funcons]
syn_pats = "pattern-list" ⟨:=⟩ multiple1 syn_pat
```

```
syn_expr :: SymbExpr Token Funcons
syn_expr = "expr"
  ⟨::=⟩ (syn_match ⟨||⟩ syn_where ⟨||⟩ syn_let ⟨||⟩ syn_functions ⟨||⟩ syn_try)
  ⟨||⟩ syn_seq
  ⟨||⟩ syn_ite
  ⟨||⟩ (syn_arr_mod ⟨||⟩ syn_str_mod ⟨||⟩ syn_rec_mod ⟨||⟩ syn_assign ⟨||⟩ syn_mutate)
  ⟨||⟩ syn_tuple
  ⟨||⟩ syn_operators
  ⟨||⟩ (syn_cons_application ⟨||⟩ syn_application)
  ⟨||⟩ (syn_arr_acc ⟨||⟩ syn_str_acc ⟨||⟩ syn_rec_acc)
  ⟨||⟩ syn_deref
  ⟨||⟩ (syn_var_expr ⟨||⟩ syn_list ⟨||⟩ syn_array ⟨||⟩ syn_record ⟨||⟩
        syn_const ⟨||⟩ syn_grouped ⟨||⟩ syn_while ⟨||⟩ syn_for)
```

Figure 12.3: The syntax of Caml Light expressions.

12.7 Expressions

Caml Light expressions have many forms. The syntax of expressions is defined via an abstract, highly ambiguous context-free grammar in [Leroy, 1997]. Our syntax description, given in Figure 12.3, corresponds closely to this grammar[4].

12.7.1 Simple Expressions

Constant expressions yield their constant as a result.

A variable within an expression evaluates to the value bound to the variable.

```
syn_var_expr = sem_var ⟨$$⟩ syn_var
sem_var s    = current_value_ [bound_ [s]]   -- implicit dereferencing of mutables
```

Parentheses group expressions, as well as the keywords **begin** and **end**. A parenthesised expression may contain a type annotation, which is ignored in the dynamic semantics.

```
syn_grouped =    parens (syn_expr ⟨** optional (keyword ":" **⟩ syn_tyexpr))
              ⟨||⟩ within (keyword "begin") syn_expr (keyword "end")
```

12.7.2 Functions

Function values come in two forms, defined via **fun** and **function**:

[4]Our description relies heavily on ambiguity reduction combinators not defined in this thesis. For an explanation of these combinators, we refer to the online documentation of the `gll` package.

```
syn_functions =     sem_fun       ⟨$$ keyword "fun" ⟨**⟩ syn_mmatch
                ⟨||⟩ sem_function ⟨$$ keyword "function" ⟨**⟩ syn_smatch
syn_mmatch :: SymbExpr Token [(Int, Funcons)]
syn_mmatch = "multiple-matching" ⟨:=⟩ longest_match (
  optional (keyword "|") **⟩ someSepBy1 syn_mcase (keyword "|"))
  where syn_mcase = sem_mcase ⟨$$⟩ syn_pats ⟨**⟩ guard ⟨** keyword "->" ⟨**⟩ syn_expr
syn_smatch :: SymbExpr Token [Funcons]
syn_smatch = "simple-matching" ⟨:=⟩ longest_match (
  optional (keyword "|") **⟩ someSepBy1 syn_scase (keyword "|"))
  where syn_scase = sem_scase ⟨$$⟩ syn_pat ⟨**⟩ guard ⟨** keyword "->" ⟨**⟩ syn_expr
guard :: SymbExpr Token (Maybe Funcons)
guard = "guard" ⟨:=⟩ optional (keyword "when" **⟩ syn_expr)
```

The only difference is that the simpler, specialised **function** disambiguates variant patterns differently. Consider the following program:

```
1 type tree = Leaf of int | Bin of tree * tree;;
2
3 let rec select = function Leaf x      -> x
4                          | Bin (l,r)  -> select l;;
```

The identifier `select` is bound to a function which receives one argument; in other words, `Leaf x` and `Bin (l,r)` are both considered a single pattern. If the function was defined with **fun** instead, `Leaf` would be parsed as a constant pattern, rendering the program invalid as `Leaf` is declared as a non-constant constructor. Semantically, **fun** is a generalisation of **function**.

```
sem_fun :: [(Int   -- number of patterns in each case alternative
            ,Funcons)] → Funcons
sem_fun cases     | n ≡ 1      = abs
                  | otherwise = curry_n_ [int_ n, abs]   -- curry_n_ is caml light specific,
                                                         -- it curries an n-ary function
  where abs = function_ [closure_ [else_cases (map snd cases) sem_match_failure]]
        n   = head (map fst cases)   -- alternatives have the same number of patterns
sem_function scases = sem_fun (map (λx → (1, x)) scases)   -- each case has 1 pattern
sem_scase p mg e = case_match_ [p, cont]
  where cont = case mg of Just cond → sequential_ [check_true_ [cond], e]
                          Nothing   → e
sem_mcase ps mg e = (length ps, sem_scase (tupler ps) mg e)
  where tupler  | length ps ≡ 1 = head
                | otherwise     = tuple_
else_cases cases def = else_ (cases ++ [def])
sem_match_failure = throw_ [variant_ [string_ "Match_failure"
                                     ,tuple_ [string_ "", int_ 0, int_ 0]]]
```

Top-level pattern alternatives may have 'guards' of the form **when** *cond* associated with them, where the condition *cond* is a Boolean expression. After a value v has successfully been matched to the pattern p of the pattern alternative, the condition of the pattern alternative is evaluated in an environment in which the bindings produced by matching v against p are active. If the condition evaluates to `false`, v is considered not to match the the pattern alternative. If the condition evaluates to `true`, the result of matching v to p is the result of matching v against the pattern alternative.

Function application is denoted by juxtaposition of expressions (left-associative):

```
syn_application = sem_apply ⟨$$⟩ syn_expr ⟨**⟩⟩ syn_expr
sem_apply e1 e2 = apply_ [e1, e2]
```

12.7.3 Local Definitions

Local definitions introduce bindings with a limited lifetime. Local definitions are introduced with the keywords **let** and **let rec**, and their scope is restricted to the expression given in an **in**-clause. Several definitions can be introduced by a single **let** -expression, separating the definitions with **and**.

```
syn_let = sem_let ⟨$$ keyword "let" ⟨**⟩ syn_rec ⟨**⟩ syn_local_defs
                  ⟨** keyword "in" ⟨**⟩ syn_expr
syn_rec :: SymbExpr Token Bool
syn_rec = "maybe-rec" ⟨:=⟩ maybe False (const True) ⟨$$⟩ optional (keyword "rec")
syn_local_defs :: SymbExpr Token ([Funcons], Funcons)
syn_local_defs = "multiple-let-bindings" ⟨:=⟩
  sem_letbs ⟨$$⟩ multipleSepBy1 syn_local_def (keyword "and")
syn_local_def :: SymbExpr Token ([Funcons], Funcons)
syn_local_def = "single-let-binding"
    -- special prioritised case for recursive bindings
  ⟨:= sem_let_var ⟨$$⟩ syn_recursive ⟨** keyword "=" ⟨**⟩ syn_expr
  ⟨||⟩ sem_let_pat ⟨$$⟩ syn_pat ⟨** keyword "=" ⟨**⟩ syn_expr
  ⟨||⟩ sem_let_abs ⟨$$⟩ syn_var ⟨**⟩ syn_pats ⟨** keyword "=" ⟨**⟩ syn_expr
```

A local definition of the form $p = e$, with p a pattern and e an expression, evaluates e and matches the result value against the pattern p. If the match is unsuccessful, a `Match_failure` exception is thrown. Otherwise, any resulting bindings are made active locally.

```
sem_let_var i e = ([i], bind_ [i, e])
sem_let_pat p e = ([], else_cases [match_ [e, p]] sem_match_failure)
sem_letbs lbs   = (concatMap fst lbs, map_unite_ (map snd lbs))
```

```
sem_let rec lb e = scope_ [lb', e]
  where lb' | rec, length (fst lb) > 0 = recursive_ [set_ (fst lb), snd lb]
            | otherwise               = snd lb
```

A local definition $p = e$ may be recursive so that bindings produced by matching the value of e to p are active when e is evaluated. This behaviour is only defined if p is an identifier or a parenthesised identifier (perhaps with type annotation), and if e evaluates to a function.

```
syn_recursive :: SymbExpr Token Funcons
syn_recursive = "rec-pat-forms"
  ⟨::=⟩ string_ ⟨$$⟩ id_lit ⟨** optional (keyword ":" ⟨** syn_tyexpr)
   ⟨||⟩ parens (syn_recursive)
```

Local definitions occurring within a single occurrence of **let rec** are mutually recursive.

A local definition may also be of the form $var\ pat_1\ \ldots\ pat_n = expr$. This form is syntactic sugar for $var =$ **fun** $pat_1\ \ldots\ pat_n \to expr$.

```
sem_let_abs v ps e = sem_let_var v (sem_fun [sem_mcase ps Nothing e])
```

The expression *expr* **where** *local_def* is syntactic sugar for **let** *local_def* **in** *expr* and *expr* **where rec** *local_def* for **let rec** *local_def* **in** *expr*:

```
syn_where = sem_where ⟨$$⟩ syn_expr ⟨** keyword "where" ⟨**⟩ syn_rec ⟨**⟩ syn_local_def
sem_where e rec lb = sem_let rec lb e
```

12.7.4 Control Operators

The expression $expr_1; \ldots; expr_n$ is evaluated by evaluating the expressions $expr_1, \ldots, expr_n$ from left to right, returning the value of $expr_n$ as a result and ignoring the values of the other expressions.

```
syn_seq = sem_seq ⟨$$⟩ multipleSepBy2 syn_expr (keyword ";") ⟨** optional (keyword ";")
sem_seq es = sequential_ $ map (λx → effect_ [x]) (init es) ++ [last es]
```

The expression **if** $expr_1$ **then** $expr_2$ **else** $expr_3$ evaluates the Boolean expression $expr_1$, and if the value of $expr_1$ equals `true`, then evaluates to the value of $expr_2$, otherwise the value of $expr_3$. The **else**-branch is optional, and defaults to **else**() if omitted.

```
syn_ite = sem_ite ⟨$$ keyword "if"  ⟨**⟩  syn_expr ⟨** keyword "then" ⟨**⟩ syn_expr
                                    ⟨**⟩⟩ optional (keyword "else" **⟩ syn_expr)
sem_ite e1 e2 me3 = if_true_else_ [e1, e2, maybe null_ id me3]
```

Using **match**, it is possible to perform case analysis within an expression:

```
syn_match =
  sem_match ⟨$$ keyword "match" ⟨**⟩ syn_expr ⟨** keyword "with" ⟨**⟩ syn_smatch
sem_match e scases = give_ [e, else_cases scases sem_match_failure]
```

Looping Caml Light has two looping constructs: **while** and **for**. A **while**-loop is written as **while** $expr_1$ **do** $expr_2$ **done** and is evaluated by evaluating $expr_2$ as long as $expr_1$ evaluates to true, evaluating $expr_1$ before each iteration.

```
syn_while = sem_while ⟨$$ keyword "while" ⟨**⟩ syn_expr ⟨** keyword "do"
                                          ⟨**⟩ syn_expr ⟨** keyword "done"
sem_while e1 e2 = while_ [e1, effect_ [e2]]
```

A **for**-loop is written as **for** *ident* = $expr_1$ **to** $expr_2$ **do** $expr_3$ **done**, or as **for** *ident* = $expr_1$ **downto** $expr_2$ **do** $expr_3$ **done** and evaluates expressions $expr_1$ and $expr_2$ to integers l and r. In the case of **to**, $expr_3$ is evaluated $r - l + 1$ times in a context in which *ident* is successively bound to the integers $l, l + 1, \ldots, r - 1, r$. In the case of **downto**, $expr_3$ is evaluated $l - r + 1$ times in a context in which *ident* is successively bound to the integers $l, l - 1, \ldots, r + 1, r$.

```
syn_for =
  sem_for ⟨$$ keyword "for" ⟨**⟩ id_lit ⟨** keyword "=" ⟨**⟩ syn_expr
          ⟨**⟩ syn_to ⟨**⟩ syn_expr ⟨** keyword "do" ⟨**⟩ syn_expr ⟨** keyword "done"
    where syn_to = "dir" ⟨:=⟩ True ⟨$$ keyword "to" ⟨|||⟩ False ⟨$$ keyword "downto"
sem_for i e1 asc e2 e3 =
  effect_ [left_to_right_map_ [case_match_ [pattern_bind_ [string_ i], e3], seq]]
    where seq | asc         = integer_sequence_ [e1, e2]
              | otherwise   = reverse_ [integer_sequence_ [e2, e1]]
```

Exceptions An expression of the form **try** *expr* **with** *smatch* evaluates *expr* and returns its value if it evaluates normally. If *expr* raises an exception value v, this value is matched against the patterns of the cases in *smatch*. The first case alternative $pat \to expr_i$ such that the exception value matches *pat* is selected and $expr_i$ evaluated. The bindings resulting from matching v against *pat* are active when $expr_i$ is evaluated. The value of $expr_i$ is the value of the **try**-expression. If no case in *smatch* is selected — i.e. no pattern is successfully matched — the exception value v propagates.

```
syn_try = sem_try ⟨$$ keyword "try" ⟨**⟩ syn_expr ⟨** keyword "with" ⟨**⟩ syn_smatch
```

```
sem_try e cases = handle_thrown_ [e, else_cases cases (throw_ [given_])]
```

Exception values can be defined by the programmer (see Section 12.8) and raised by applying the operation `raise` from the core library to an exception value.

12.7.5 Operations on Data

Tuples An expression $expr_1, \ldots, expr_n$ evaluates to an n-tuple holding the values of the expressions in order. The order in which the expressions are evaluated is unspecified.

```
syn_tuple      = shortest_match (sem_tuple ⟨$$⟩ multipleSepBy2 syn_expr (keyword ","))
sem_tuple es = tuple_ es
```

Variants An expression *ncconstr expr* applies the function value to which the global name *ncconstr* is bound to the value of *expr* (which may be a tuple). The binding of *ncconstr* is introduced by a type definition (see Section 12.8). When applied, the function value constructs a variant value with one or more mutable values as arguments (tupled, if more than one).

```
syn_cons_application = sem_unary_app ⟨$$⟩ syn_ncconstr ⟨**⟩ syn_expr
sem_unary_app v e1   = sem_apply (bound_ [v]) e1
```

Lists The expression $[expr_1; \ldots; expr_n]$ is equivalent to $expr_1 :: \ldots :: expr_n :: []$. If the behaviour of :: is modified, then so is the behaviour of list notation.

```
syn_list = shortest_match (sem_list ⟨$$⟩
  brackets (multipleSepBy1 syn_expr (keyword ";") ⟨** optional (keyword ";")))
sem_list = foldr (flip sem_cons "::") (list_ [])   -- see Section 12.7 for sem_cons
```

Arrays The expression $[|\ expr_1; \ldots; expr_n\ |]$ evaluates to an array of n mutable values. The array has indices $0, \ldots, n-1$ and the mutable value at position $i-1$ is the value of $expr_i$ initially, for all $1 \leqslant i \leqslant n$. The order in which the expressions are evaluated is unspecified.

```
syn_array = shortest_match (sem_array ⟨$$⟩ within
              (keyword "[|")
                (multipleSepBy1 syn_expr (keyword ";") ⟨** optional (keyword ";"))
              (keyword "|]"))
sem_array es = vector_ (map mkVecElem es)
  where mkVecElem e = allocate_initialised_variable_ [values_, e]
```

The expression $expr_1$.($expr_2$) is equivalent to `vect_item` $expr_1$ $expr_2$. Similarly, the expression $expr_1$.($expr_2$) ← $expr_3$ is equivalent to `vect_assign` $expr_1$ $expr_2$ $expr_3$. The operations `vect_item` and `vect_assign` are part of the core library [Leroy, 1997]. If the programmer changes the behaviour of `vect_item` and `vect_assign`, the behaviour of $expr_1$.($expr_2$) and $expr_1$.($expr_2$) ← $expr_3$ changes accordingly.

```
syn_arr_acc  = sem_arr_acc  ⟨$$⟩ syn_expr ⟨** keyword ".(" ⟨**⟩ syn_expr ⟨** keychar ')'
syn_arr_mod = sem_arr_mod ⟨$$⟩ syn_expr ⟨** keyword ".(" ⟨**⟩ syn_expr ⟨** keychar ')'
                    ⟨⟨⟨** keyword "<-" ⟨**⟩ syn_expr
sem_arr_acc e1 e2 = sem_apply (sem_unary_app (string_ "vect_item") e1) e2
sem_arr_mod e1 e2 e3 =
  sem_apply (sem_apply (sem_unary_app (string_ "vect_assign") e1) e2) e3
```

Strings A string is a sequence of mutable characters. The mutable characters of a string are accessed and modified similarly to array elements. The expression $expr_1$.[$expr_2$] is equivalent to `nth_char` $expr_1$ $expr_2$ and $expr_1$.[$expr_2$] ← $expr_3$ is equivalent to `set_nth_char` $expr_1$ $expr_2$ $expr_3$.

```
syn_str_acc  = sem_str_acc  ⟨$$⟩ syn_expr ⟨** keyword ".[" ⟨**⟩ syn_expr ⟨** keychar ']'
syn_str_mod = sem_str_mod ⟨$$⟩ syn_expr ⟨** keyword ".[" ⟨**⟩ syn_expr ⟨** keychar ']'
                          ⟨⟨⟨** keyword "<-" ⟨**⟩ syn_expr
sem_str_acc e1 e2 = sem_apply (sem_unary_app (string_ "nth_char") e1) e2
sem_str_mod e1 e2 e3 =
  sem_apply (sem_apply (sem_unary_app (string_ "set_nth_char") e1) e2) e3
```

Records The expression $\{label_1 = expr_1, \ldots, label_n = expr_n\}$ evaluates to a record value in which $label_i$ maps to a mutable value, initially the value of $expr_i$, for all $1 \leqslant i \leqslant n$. The order in which the expressions are evaluated is unspecified. No label may occur twice.

```
syn_record = shortest_match (sem_rec ⟨$$⟩
  braces (multipleSepBy1 syn_entry (keyword ";") ⟨** optional (keyword ";")))
  where syn_entry = "entry" ⟨:=⟩ sem_entry ⟨$$⟩ syn_label ⟨** keyword "=" ⟨**⟩ syn_expr
        sem_entry l e = tuple_ [l, allocate_initialised_variable_ [values_, e]]
sem_rec entries = record_ [map_ entries]
```

Record fields are accessed and modified similarly to the elements of an array. However, the behaviour of these operations cannot be changed. The expression $expr_1$. *label* evaluates $expr_1$ to a record and returns the mutable value mapped to by *label* in the record. The expression $expr_1$. *label* ← $expr_2$ evaluates $expr_1$ to a record, and mutates the value mapped to by *label* in the record, and returns ().

```
syn_rec_acc  = sem_rec_acc  ⟨$$⟩ syn_expr ⟨** keyword "." ⟨**⟩ syn_label
syn_rec_mod = sem_rec_mod ⟨$$⟩
  syn_expr ⟨** keyword "." ⟨**⟩ syn_label ⟨⟨⟨** keyword "<-" ⟨**⟩ syn_expr
sem_rec_acc e1 e2      = assigned_ [record_select_ [e1, e2]]
sem_rec_mod e1 e2 e3 = assign_ [record_select_ [e1, e2], e3]
```

If identifier *ident* is bound to the mutable value, then the expression *ident* ← *expr* makes an in-place modification, changing the mutable value bound by *ident* to the value of *expr*.

```
syn_mutate        = sem_mutate ⟨$$⟩ id_lit ⟨⟨⟨** keyword "<-" ⟨**⟩ syn_expr
sem_mutate e1 e2 = assign_ [bound_ [string_ e1], e2]
```

As an example, consider the following program (**mutable** is introduced in Section 12.8):

```
1 type coord = { mutable x : int; mutable y : int };;
2
3 let origin = { x = 0; y = 0 };;
4 let move_north { x = xp } = xp <- xp + 1;;
5
6 (* the following prints { x = 2, y = 0 } *)
7 move_north origin ; move_north origin ; origin;;
```

12.7.6 Operators

The priorities and associativity of the operators in Caml Light's core library are listed in the operator-table shown in Figure 12.4. The semantics of these operators, and of functions in the core library generally, are described in Chapter 13 of [Leroy, 1997] and are not repeated here.

```
caml_light_core_library_ = ...   -- caml light specific funcon
```

Two operators do not appear in the operator-table of Figure 12.4: `!` and `:=`. This is because `!` has a higher priority than function application, which in turn has a higher priority than all other operators. Conversely, `:=` has a lower priority than `,` (tuple constructor), which in turn has a lower priority than all other operators (see Figure 12.3).

```
syn_assign = sem_infix  ⟨$$⟩ syn_expr ⟨**⟩ keyword ":=" ⟨⟨⟨**⟩ syn_expr
syn_deref  = sem_prefix ⟨$$⟩ keyword "!" ⟨**⟩ syn_expr
```

The operators `&` and `or` (and their synonyms `&&` and `||`) are special because they are not strict. The expression $expr_1$ `&` $expr_2$ is equivalent to **if** $expr_1$ **then** $expr_2$ **else** `false`

```
operator_syntax = [
  (11, [("-",    Prefix sem_minus), ("-.", Prefix sem_minusf)])
  ,(10,[("**",   Infix sem_infix RAssoc)])
  ,(9,  [("mod", Infix sem_infix LAssoc)])
  ,(8,  [("*",   Infix sem_infix LAssoc), ("*.",  Infix sem_infix LAssoc)
        ,("/",   Infix sem_infix LAssoc), ("/.",  Infix sem_infix LAssoc)])
  ,(7,  [("+",   Infix sem_infix LAssoc), ("+.",  Infix sem_infix LAssoc)
        ,("-",   Infix sem_infix LAssoc), ("-.",  Infix sem_infix LAssoc)])
  ,(6,  [("::",  Infix sem_cons RAssoc)])
  ,(5,  [("@",   Infix sem_infix RAssoc), ("^",   Infix sem_infix RAssoc)])
  ,(4,  [("=",   Infix sem_infix LAssoc), ("<>",  Infix sem_infix LAssoc)
        ,("==",  Infix sem_infix LAssoc), ("!=",  Infix sem_infix LAssoc)
        ,("<",   Infix sem_infix LAssoc), ("<.",  Infix sem_infix LAssoc)
        ,("<=",  Infix sem_infix LAssoc), ("<=.", Infix sem_infix LAssoc)
        ,(">",   Infix sem_infix LAssoc), (">.",  Infix sem_infix LAssoc)
        ,(">=",  Infix sem_infix LAssoc), (">=.", Infix sem_infix LAssoc)])
  ,(3,  [("not", Prefix sem_prefix)])
  ,(2,  [("&",   Infix sem_and  LAssoc), ("&&",  Infix sem_and  LAssoc)])
  ,(1,  [("or",  Infix sem_or   LAssoc), ("||",  Infix sem_or   LAssoc)])]
syn_operators :: SymbExpr Token Funcons
syn_operators = fromOpTable "operators" (opTableFromList operator_syntax) syn_expr
sem_infix e1 op e2 = sem_apply (sem_prefix op e1) e2
sem_prefix op e1   = sem_unary_app (sem_op_var op) e1
```

Figure 12.4: Relative priorities and associativity of operators.

unless `prefix &` is redefined by the programmer. Similarly, the expression $expr_1$ `or` $expr_2$ is equivalent to **if** $expr_1$ **then** `true` **else** $expr_2$, unless `prefix or` is redefined.

```
sem_and e1 op e2 = else_ [sequential_ [effect_ [bound_ [sem_op_var op]], sem_infix e1 op e2]
                         , sem_ite e1 e2 (Just (bool_ False))]
sem_or e1 op e2  = else_ [sequential_ [effect_ [bound_ [sem_op_var op]], sem_infix e1 op e2]
                         , sem_ite e1 (bool_ True) (Just e2)]
```

The programmer can change the behaviour of & and &&, as well as the behaviour of `or` and ||, separately.

The core library defines `minus` and `minus_float` for the prefix operators $-$ and $-.$ respectively, avoiding overlap with the identifiers `prefix` $-$ and `prefix` $-.$ which are used for the infix versions of these operators:

```
sem_minus _ e1  = sem_unary_app (string_ "minus") e1
sem_minusf _ e1 = sem_unary_app (string_ "minus_float") e1
```

The operator :: is special in that it is the constructor of a variant value. Just as other

constructors, it is therefore uncurried, and receives a single pair as argument. The expression $expr_1 :: expr_2$ is therefore equivalent to `prefix` $:: (expr_1, expr_2)$.

```
sem_cons e1 op e2 = sem_prefix op (tuple_ [e1, e2])
```

12.8 Module Implementations

A module is a sequence of 'phrases', where each phrase is either a type definition, an exception definition, a value definition, a directive, or a top-level expression.

```
syn_phrase :: SymbExpr Token (Funcons → Funcons)
syn_phrase = "impl-phrase"
   ⟨:=⟩ sem_phrase_expr     ⟨$$⟩ syn_expr
   ⟨||⟩ sem_phrase_let      ⟨$$ keyword "let" ⟨**⟩ syn_rec ⟨**⟩ syn_local_defs
   ⟨||⟩ sem_global_variants ⟨$$⟩ syn_type
   ⟨||⟩ sem_global_variants ⟨$$⟩ syn_exc
   ⟨||⟩ id                  ⟨$$ keyword "#" ⟨** id_lit ⟨** string_lit   -- directives
sem_global_variants envs rest = scope_ [map_unite_ envs, rest]
```

Phrases are executed in order, and their effects may influence subsequent phrases. Directives control the behaviour of compilers and are ignored here. Top-level expressions are evaluated and the resulting value is converted to a string and printed to the standard output. After a top-level expression is executed, the next phrase is executed uninfluenced.

```
sem_phrase_expr e1 next = sequential_ [show_excs (print_ [cl_output e1]) null_, next]
  -- report any uncaught exceptions
show_excs code def = handle_thrown_ [code, sequential_ [print_ [cl_exc_output given_], def]]
  -- funcon caml_light_show_ is a language specific funcon for pretty-printing values
cl_output e1     =
  string_append_ [string_ "#- : <type> = ", caml_light_show_ [e1], string_ "\n"]
cl_exc_output e1 =
  string_append_ [string_ "Uncaught exception: ", caml_light_show_ [e1], string_ "\n"]
```

If evaluating the expression raises an exception, the exception is converted to a string and printed to the standard output — in a way which distinguishes it from a regular value — and the next phrase is executed uninfluenced.

A value definition is a **`let`** expression without an **`in`**-clause. The bindings introduced by the definition range over the subsequent phrases.

```
sem_phrase_let rec lbs next = sem_let rec (fst lbs, show_excs (snd lbs) (map_ [])) next
```

If computing the bindings raises an exception, the exception is converted to a string and printed to the standard output. In this case, the next phrase is executed without any new active bindings.

A type definition introduces a new type, which is either a variant type, a record type, or a type abbreviation. Type definitions are written following the keyword **type** and, if more than one, are separated by **and**.

```
syn_type :: SymbExpr Token [Funcons]
syn_type = "type-definition"
  ⟨:=⟩ concat ⟨$$ keyword "type" ⟨**⟩ multipleSepBy1 syn_type_def (keyword "and")
syn_type_def :: SymbExpr Token [Funcons]
syn_type_def = "typedef"
  ⟨:=⟩ id ⟨$$ syn_params ⟨** id_lit ⟨** keyword "="  ⟨**⟩ syn_decl_constrs
  ⟨||⟩  [] ⟨$$ syn_params ⟨** id_lit ⟨** keyword "="  ⟨** syn_decl_labels
  ⟨||⟩  [] ⟨$$ syn_params ⟨** id_lit ⟨** keyword "==" ⟨** syn_tyexpr
  ⟨||⟩  [] ⟨$$ syn_params ⟨** id_lit   -- abstract types
syn_params :: SymbExpr Token [String]
syn_params = "type-params"
  ⟨:=⟩ satisfy []
  ⟨||⟩ (:[]) ⟨$$ keyword "'" ⟨**⟩ id_lit
  ⟨||⟩ parens (multipleSepBy1 (keyword "'" **⟩ id_lit) (keyword ","))
syn_decl_labels :: SymbExpr Token [String]
syn_decl_labels = "label-decls" ⟨:=⟩ braces (multipleSepBy1 syn_decl_label (keyword ";"))
syn_decl_label :: SymbExpr Token String
syn_decl_label = "label-decl" ⟨:=⟩ syn_mutable **⟩ id_lit ⟨** keyword ":" ⟨** syn_tyexpr
syn_mutable = maybe False (const True) ⟨$$ optional (keyword "mutable")
```

A variant type definition describes its constructors and binds the names of its constructors to their implementation. The bindings are active during the execution of subsequent phrases.

```
syn_decl_constrs :: SymbExpr Token [Funcons]
syn_decl_constrs = "const-decls" ⟨:=⟩ multipleSepBy1 syn_decl_constr (keyword "|")
syn_decl_constr :: SymbExpr Token Funcons
syn_decl_constr = "const-decl"
  ⟨:=⟩ sem_null_constr ⟨$$⟩ alt_id_lit
  ⟨||⟩  sem_constr      ⟨$$⟩ alt_id_lit ⟨** keyword "of" ⟨** syn_mutable ⟨** syn_tyexpr
sem_null_constr i = bind_ [string_ i, variant_ [string_ i, null_]]
sem_constr i      = bind_ [string_ i, function_ [closure_ [variant_ [string_ i, args]]]]
  where args = if_true_else_
                 [is_in_type_ [given_, tuples_ [values_*]]]
                 , tuple_ [interleave_map_ [allocate_initialised_variable_ [values_, given_]
                                            , tuple_elements_ [given_]]]
                 , allocate_initialised_variable_ [values_, given_]]
```

A constructor receiving at least one argument may be applied to a single argument or to a tuple of arguments. In both cases, the arguments need to be assigned to newly allocated variables. The definition of *sem_constr* above checks whether multiple arguments are given in a tuple (with **is_in_type_**). If this is the case, **interleave_map_** is used to allocate fresh variables for all elements of the tuple. Otherwise, one fresh variable is allocated for the single given argument.

Exception values are the values of a built-in variant type `exn`. An exception definition extends this type with one or more constructors.

```
syn_exc :: SymbExpr Token [Funcons]
syn_exc = "exception-definition"
  ⟨:=⟩ keyword "exception" **⟩ multipleSepBy1 syn_decl_constr (keyword "and")
```

Record type definitions and type abbreviations introduce no bindings.

The phrases of a module are followed by a double semicolon.

```
syn_module :: SymbExpr Token Funcons
syn_module = "module" ⟨:=⟩
  sem_module ⟨$$⟩ foldr_multiple (syn_phrase ⟨** keyword ";;") null_
sem_module body = initialise [scope_ [caml_light_core_library_, body]]
  where initialise b = initialise_storing_ [initialise_binding_ [initialise_giving_ b]]
```

A module is executed by executing its phrases in order. The core library is loaded by default.

12.9 Interpretation

The following code fragment defines a main function for interpreting Caml Light programs by parsing them, translating them to funcon terms, and executing them.

```
main = do
  args ← getArgs
  case args of
    []      → putStrLn "Please provide me with an input file"
    f : opts → go f opts
where
  go :: FilePath → [String] → IO ()
  go f opts = do
    str ← readFile f
    let tokens = lexer str
    let fcts   = parse syn_module tokens
    if (null fcts)
```

```
then putStrLn "no parse"
else  runWithExtensions CL.funcons CL.entities CL.types opts (Just (head fcts))
```

The function *lexer* :: *String* → [*Tokens*] is imported from the module `Lexer` (not given). The Caml Light specific funcons are exported by the module `Funcons.CamlLight.Library`, which is imported under the qualified name *CL*.

Chapter 13

Tool Evaluation

References *The material in Section 13.1 is taken from [Van Binsbergen et al., 2018]. The data on Caml Light differs slightly because the syntax description of Caml Light in this thesis is a modified version of the description used to collect the data in the paper.*

In this chapter we investigate the runtime efficiency of the main tools of the `gll` and `funcons-tools` packages (Section 13.1 and Section 13.2) and reflect on using these tools in the case studies of the previous chapters (Section 13.3).

13.1 Parsing with BNF Combinators

In this section we evaluate the efficiency of the BNF combinators provided by the `gll` package. Compared to `gll`, the code fragments of Chapter 5 omit several features that improve usability. For example, the implementation of FUN-GLL in `gll` can produce BSPPFs, not just EPNs, and throws error messages to aid debugging. Most significant are the omission of lookahead and ambiguity reduction. Ambiguity reduction strategies, such as longest match, are implemented in variations of $\langle ::= \rangle$, $\langle ||| \rangle$, and $\langle ** \rangle$ by filtering the pivots extracted from an EPN set during the semantic phase. Configuration options enable ambiguity reduction strategies globally. Lookahead is performed in the way described by [Scott and Johnstone, 2013]: functions *descend* and *ascend* yield only the descriptor $((x, \alpha, \beta), l, k)$ if the k-th terminal of the input is in the lookahead-set pre-computed for

(x, α, β).

The GLL package exports two modules — `GLL.Combinators.Interface` and `GLL.Combinators.BinaryInterface` — providing BNF combinators that are very similar superficially, but differ significantly internally. The core combinators of `BinaryInterface` specialise the core combinators of `Interface`, converting all expressions to symbol expressions as in §5.3.4. The grammars generated by the underlying grammar combinator expressions are binarised in the same sense as P3's internal grammars [Ridge, 2014]. The two modules are often interchangeable, as the combinators of `Interface` have more general types than those of `BinaryInterface`. Specifically, when a syntax description is originally written with the combinators of `BinaryInterface`, one can change the import of `BinaryInterface` to `Interface` and the syntax description is still valid. In this section we take advantage of this fact and evaluate the two modules, demonstrating the effects of grammar binarisation and lookahead on running FUN-GLL.

Parser evaluation The syntax descriptions of three software languages written with the BNF combinators are evaluated: ANSI-C, Caml Light, and CBS. The syntax description of Caml Light is given in Chapter 12. The syntax description of CBS has been taken from the CBS compiler (package `funcons-intgen`). The syntax description of ANSI-C has been generated and is based on the grammar originally given in [Kernighan and Ritchie, 1988]. The running times include a lexicalisation phase which produces a sequence of tokens (values of type *Token*, see §5.1.1), given as input to the parsing phase. For each language we selected considerable software-language engineering projects: a parser generator in ANSI C, a Caml Light compiler in Caml Light, and a complete semantic specification in CBS. The test files are the result of composing varying selections of source files taken from these projects. The tests have been executed on a laptop with quad-core 2.4GHz processors and 8GiB of RAM, under Ubuntu 14.04.

The data for ANSI-C is given in Table 13.1 and is visualised in Figure 13.1. The syntax description is a direct transcription of the grammar listed in [Kernighan and Ritchie, 1988], which is written in BNF without extensions. The grammar is nondeterministic and left-recursive. The internal grammar given to FUN-GLL has 229 alternates and 71 nonterminals. When written with `BinaryInterface`, a large number of alternates and nonterminals is

Tokens	1515	8411	15589	26551	36827
Flexible	0.44	2.46	4.83	7.86	10.40
+lookahead	0.50	2.77	4.75	7.23	10.27
Binarised	1.12	7.17	13.47	22.47	32.41
+lookahead	1.31	6.67	12.02	18.9	25.30

Table 13.1: Parsing ANSI-C files (in seconds).

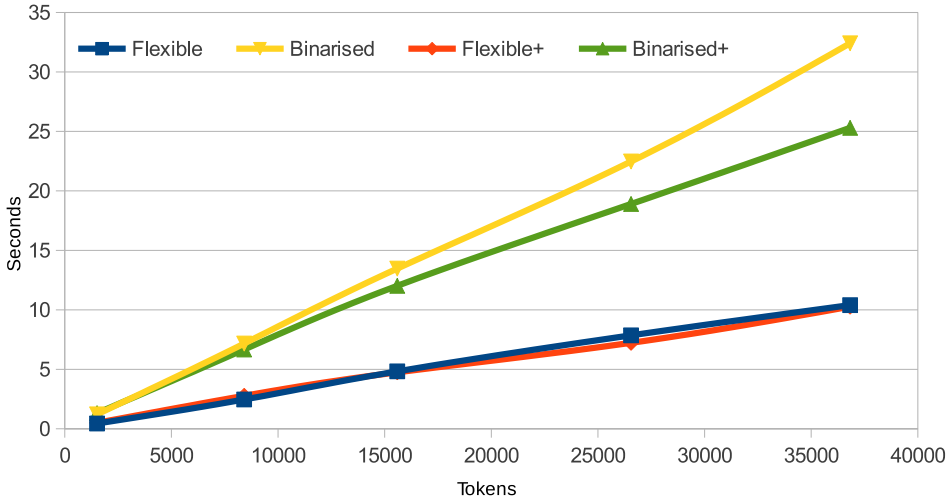


Figure 13.1: Visualisation of the data in Table 13.1.

generated: the internal grammar has 848 alternates and 690 nonterminals. The running times of FUN-GLL are strongly affected by the binarisation of the grammar, differing with a factor between 2.4 and 2.6 with lookahead and between 2.5 and 3.1 without lookahead. Lookahead only has a significant effect when the grammar is binarised.

The data in Table 13.2 (visualised in Figure 13.2) shows the running times for Caml Light. The syntax description is that of Chapter 12, which is based on the grammar of the Caml Light reference manual [Leroy, 1997]. The grammar is highly nondeterministic and has many sources of ambiguity. In particular, the grammar contains a large and highly nondeterministic nonterminal for expressions. The combinator description of the grammar makes heavy use of abstraction to capture EBNF notations, and some coercions between combinator expressions are necessary when written with `Interface`. The difference in size between the grammars given to FUN-GLL are therefore smaller: 300 versus 694 alternates

Tokens	1097	2813	4534	8846	15910	28703
Flexible	1.69	3.23	5.09	11.51	17.52	35.94
+lookahead	1.74	3.21	5.32	10.87	17.84	33.08
Binarised	2.07	4.18	6.90	15.40	25.59	49.79
+lookahead	2.06	3.90	6.50	13.06	21.56	39.62

Table 13.2: Parsing Caml Light files (in seconds).

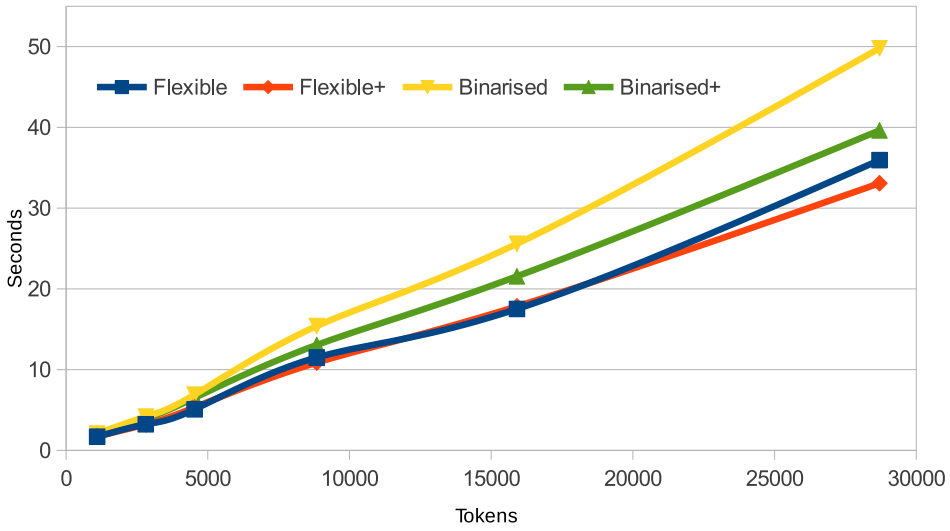


Figure 13.2: Visualisation of the data in Table 13.2.

and 139 versus 533 nonterminals. The negative effect of grammar binarisation is also smaller, around 1.2 with lookahead and between 1.22 and 1.46 without lookahead.

Evaluating the semantic phase The data in Tables 13.1 and 13.2 exclude the semantic phase. The data in Table 13.3 (visualised in Figure 13.3) shows the running times of parsing CBS files, applying the semantics phase with disambiguation, and pretty-printing the resulting abstract syntax tree. The effect of binarising the grammar is not as big as in the case of ANSI-C: between 1.7 and 2.1 with lookahead (without lookahead omitted). The grammar given to FUN-GLL by `Interface` has 257 alternates and 126 nonterminals, versus 771 alternates and 640 nonterminals by `BinaryInterface`. The effect of lookahead is dramatic and lookahead is required to keep the running times under control as the input grows.

Tokens	2653	14824	17593	21162	26016
Flexible	1.36	12.01	15.24	20.17	29.54
+lookahead	1.10	5.56	6.54	7.86	9.57
Binarised	2.94	41.83	–	–	–
+lookahead	2.31	9.51	11.38	13.48	16.62

Table 13.3: Parsing and pretty-printing CBS files (in seconds).

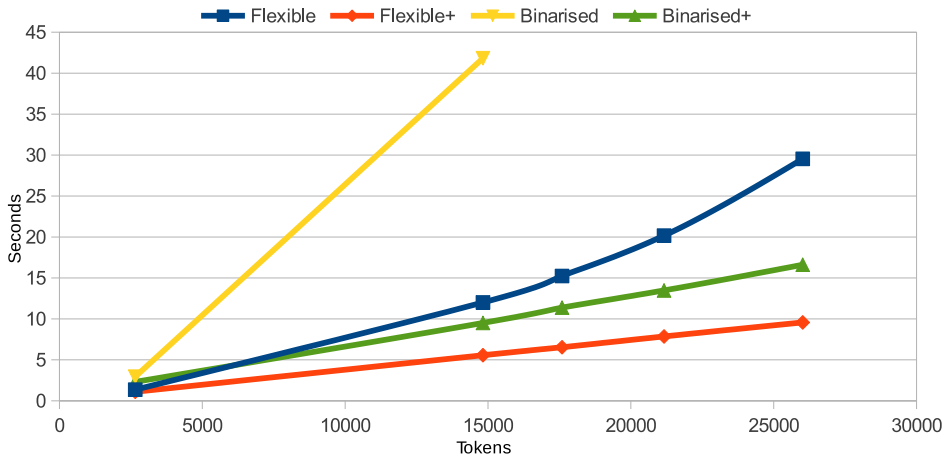


Figure 13.3: Visualisation of the data in Table 13.3.

The syntax descriptions used in this evaluation have not been refactored to enhance parser performance. Running times will decrease when the grammars are rewritten to avoid, for example, nonterminal generation (when using `Interface`) and nondeterminism.

Constant overhead The running times include a constant component depending[1] on the size of the grammar, because the internal grammar, and lookahead sets for the grammar, are computed the first time a particular syntax description is used for parsing. This is a waste when a compiler or interpreter does not use the same syntax description a second time (without being restarted itself). Duregård and Jansson have developed an 'embedded parser generator' library using meta-programming, in which Template Haskell fragments define a grammar for which a parser is generated at compiletime [Duregård and Jansson, 2011]. Similarly, Devriese and Piessens have used Template Haskell to perform grammar transformation on the grammars generated by their combinators at compiletime [Devriese and Piessens, 2011].

[1] But crucially independent of the size of the input sentence.

We expect these techniques are applicable to the BNF combinators so that grammar generation and lookahead computation can be performed at compiletime, thus avoiding the constant overhead.

13.2 Interpreting Funcon Terms

The funcon term interpreters of the Haskell Funcon Framework are used to verify CBS specifications [Van Binsbergen et al., 2016, Van Binsbergen et al., 2019], replacing the Prolog interpreters previously used by the PLANCOMPS project [Bach Poulsen and Mosses, 2014b]. The interpreters have been developed to gain confidence in the correctness of language specifications by executing programs that test specific aspects of the specified language. This section provides empirical data demonstrating that the funcon term interpreters are fit for purpose.

The Haskell Funcon Framework has been used to test the CBS specifications of the example languages provided as part of the beta release of CBS (IMP, Simple, MiniJava, SL, and OCaml Light), the case studies of this thesis, and the CBS specification of a significant subset of C#. The experience with these languages has shown that:

- Specifications can be tested for the most part with programs that execute in a matter of seconds. However, specific programs may prove difficult for reasons related to the specifics of individual funcon definitions
- Refocussing is crucial and speeds up the execution of a large variety of programs. The impact of refocussing is most significant on programs that involve iteration and nested function calls, making it possible to execute such programs with larger inputs
- The backtracking approach may undo the results of potentially costly rewrites and computations as part of rule selection. The order in which rules are considered can therefore significantly impact performance

To confirm the usability of the tools, we include Table 13.4, which has been taken from [Van Binsbergen et al., 2019]. The table compares the funcon term interpreter of the Simple language with the $\mathbb{K}$ tools of the $\mathbb{K}$ framework (in which Simple was originally defined), by running a number of test programs that come bundled with distributions of

Test Program	Funcon Term Interpreter		$\mathbb{K}$ tool
	Unoptimised	Refocussing Enabled	
`exception tests 1 to 15`	11.4	1.67	30.3
`div-nondet`	0.2	0.06	1.9
`factorial`	2.8	0.18	1.9
`collatz`	11.3	0.53	1.9
running total	25.6	2.43	35.9
`higher-order`	-	3.02	10.1
`matrix`	-	5.84	2.2
`sortings`	-	6.11	2.9
running total	-	17.4	51.1

Table 13.4: Running times in seconds of the Simple funcon interpreter and $\mathbb{K}$ tool.

the $\mathbb{K}$ tool [Lazar et al., 2012, The $\mathbb{K}$ Framework, 2018]. These results were produced on a virtual private server with four 2.2GHz virtual CPUs and 8GiB of RAM, under Ubuntu 18.04. The Haskell funcon modules were compiled using GHC 8.0.2. The $\mathbb{K}$ specification of Simple was compiled with the Java back end of version 5 of the $\mathbb{K}$ tool and loaded in server-mode to avoid compilation overhead on each run. Several runs of 'JVM warm up' have been performed. The numbers are the average runtimes of 10 runs.

Without refocussing enabled, the tests `higher-order`, `matrix` and `sortings` cause the memory to overflow. These tests involve a relatively large number of function calls or loop-iterations, which result in considerable growth of the funcon term under evaluation, thus increasing the overhead of decomposing and recomposing the term at each step of the small-step evaluation. Decomposition involves potentially a lot of backtracking and undoing context-free rewrites unnecessarily, as discussed in §9.2.1. With refocussing, these programs execute within two to four seconds.

The data for Simple is exemplary for the behaviour of funcon term interpreters on the test programs of the aforementioned languages. In the next subsections we focus on the effects of rule orderings and refocussing. All tests that follow have been executed on a laptop with quad-core 2.4GHz processors and 8GiB of RAM, under Ubuntu 14.04 with GHC 8.2.1. The tests are performed with refocussing enabled, unless otherwise specified.

13.2.1 The Costs of Backtracking

The funcon translation of Mini is such that every procedure invocation involves **handle-return** (although in principle **handle-return** is redundant under some circumstances, e.g. if

a procedure has no return statements). As this section demonstrates, the implementation of **handle-return** has a tremendous influence on the efficiency with which procedure invocations are evaluated. Consider its definition:

$$\textit{Funcon}\ \textbf{handle-return}(\_ :\Rightarrow T) :\Rightarrow T \tag{13.1}$$

$$\textit{Rule}\ \frac{X \xrightarrow{\text{abrupted}()} X'}{\textbf{handle-return}(X) \xrightarrow{\text{abrupted}()} \textbf{handle-return}(X')} \tag{13.2}$$

$$\textit{Rule}\ \frac{X \xrightarrow{\text{abrupted}(\textbf{returned}(V:\textbf{values}))} X'}{\textbf{handle-return}(X) \xrightarrow{\text{abrupted}()} V} \tag{13.3}$$

$$\textit{Rule}\ \frac{X \xrightarrow{\text{abrupted}(V':\sim\textbf{returning})} X'}{\textbf{handle-return}(X) \xrightarrow{\text{abrupted}(V')} X'} \tag{13.4}$$

$$\textit{Rule}\ \textbf{handle-return}(V : T) \rightsquigarrow V \tag{13.5}$$

Each computational rule is only applicable if there is a $\longrightarrow$-transition on X. Which of the computational rules is applicable depends on the presence of an abrupted signal and, if there is a signal, whether it is a signal of type **returning**. The funcon term interpreter may transition X three times to determine which rule is applicable, discarding the result of each unsuccessful attempt by backtracking. The order in which these rules are considered makes a big difference, because X may be an expensive computation, which we demonstrate with an example. The following Mini programs print the greatest common divisor (GCD):

```
1 procedure gcd (var p, var q) begin
2         if (p == q) then return p;
3   else if (p < q)  then return (gcd (q, p));
4   else                  return (gcd (p-q,q));
5 end
6 print (gcd(22,8));
```

```
1 procedure gcdp(var p, var q) begin
2         if (p == q) then print p;
3   else if (p < q)  then gcdp (q, p);
4   else                  gcdp (p-q,q);
5 end
6 gcdp(22,8);
```

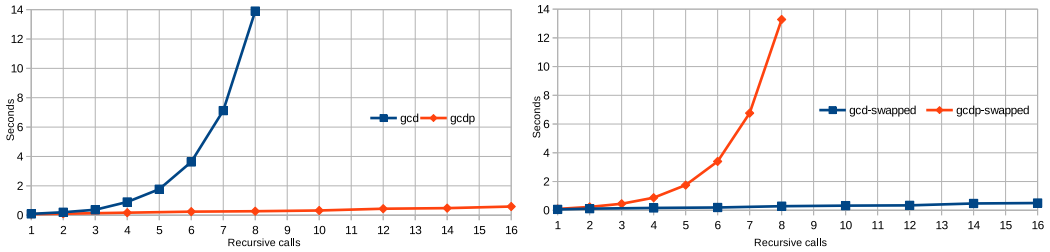


Figure 13.4: Running times of GCD programs with different rule order for **handle-return**. The x-axis shows the number of recursive calls.

The second program (gcdp) prints the GCD as soon as it is found, whereas the first program (gcd) returns the GCD for subsequent printing. Running times for these two programs are given in Figure 13.4. If the rules of **handle-return** are considered in the order they are written above, then gcd exhibits exponential running times in the number of recursive calls, whereas gcdp exhibits small, linear growth (left-hand side of Figure 13.4). If rules (13.2) and (13.3) are swapped then, the growth in running times of the two programs is reversed (right-hand side of Figure 13.4). This experiment shows that the details of funcon definitions may have tremendous impact on the efficiency of the funcon interpreters generated from them. Moreover, the experiment shows that it is not possible to determine the optimal rule order by inspecting the rules alone.

Besides premises, backtracking also undoes the work of any side-conditions that may have been executed. As a result, many rewrites may be undone. However, because rewriting is context independent, there is never a need to undo rewrites. It would therefore be beneficial to somehow memoise the results of rewrites, for example by maintaining the funcon term in a mutable data structure and letting rewrites mutate the funcon term directly.

In general, the negative impact of backtracking can be reduced by 'left-factoring': merging the common parts of rules and pushing the point to which backtracking reverts forward [Pettersson, 1999]. In the case of **handle-return**, this involves separating: (1) performing the transition from X to X' and (2) matching any abrupted signal that emerges from that transition against a pattern. When separated, the work involved with (1) can be shared across rule implementations so that backtracking only undoes the work involved with (2). The translation of CBS rules to IML, discussed in Chapter 8, separates premises in this way. Moreover, the CBS to IML translation makes rewriting explicit. The IML definitions

of funcons are therefore a suitable target for left-factoring.

13.2.2 Dynamic Refocussing

This subsection looks at the effects of refocussing, confirming that the implementation of dynamic refocussing in the Haskell Funcon Framework has similar effects as originally reported for the Prolog interpreters [Bach Poulsen and Mosses, 2014b].

This experiments involves three Caml Light programs: another recursive GCD program, a recursive Fibonacci program, and an iterative factorial program:

```
let rec gcd = fun p q ->
                  if p == q then p
            else  if p <  q then gcd q p
            else                 gcd (p-q) q;;
gcd 230 178;;
```

```
let rec fib = fun 0 -> 0
                | 1 -> 1
                | x -> fib (x-1) + fib (x-2);;
fib 11;;
```

```
let prod = ref 1;;
for i = 1 to 17 do prod := !prod * i done;;
!prod;;
```

The running times of these programs with and without refocussing are shown in Figure 13.5. The GCD program has been tested with inputs that result in one to 17 function calls, the Fibonacci program with inputs 1 to 11, and the factorial program with one to 17 iterations.

Without refocussing, the GCD and Fibonacci program show incredible growth in running times, making it impossible to execute these programs with non-trivial input. The running times of the Factorial program grow linearly with about a second per additional iteration. These runtimes are heavily affected by the large core library of Caml Light, which appears as an environment in the first argument of **scope** in the funcon terms generated for these programs. The first argument of **scope** is type-checked to ensure that it is of type **environments** (see Rules (8.93) and (8.94) on page 168). Type-checking maps involves decomposing the map and type-checking all its keys and values. Without refocussing, this needs to happen repeatedly for each occurrence of **scope**, and backtracking undoes the action when selecting between the rules of **scope**.

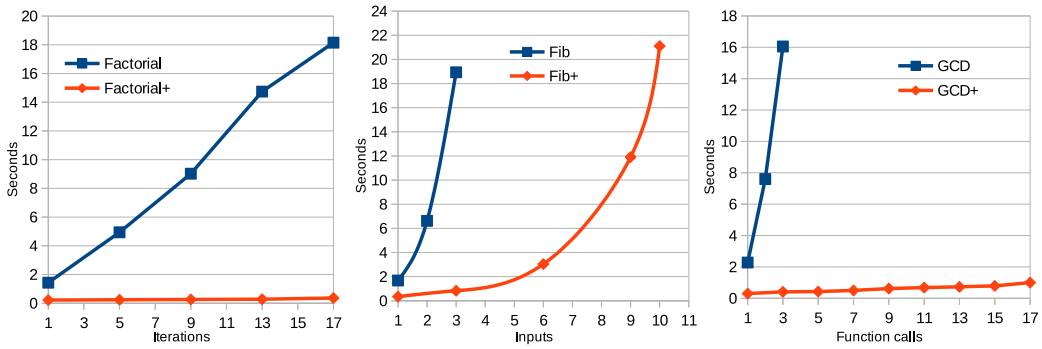


Figure 13.5: Three Caml Light programs with and without dynamic refocussing.

With refocussing, the Fibonacci program grows exponentially, as expected, at a much lower rate than without refocussing. Particularly positive is that the growth-rate is not greater than the golden ratio, i.e. the funcon term interpreter does not add to the worst-case complexity of the program, which is $O(1.618^n)$. With refocussing, both the GCD and factorial program grow linearly at a very slow pace.

13.3 Case Study Evaluation

This section reflects on the formal descriptions of syntax and semantics in the Mini and Caml Light case studies specifically, and the usability of the tools in general.

13.3.1 BNF Combinators

In our experience, the BNF combinators are practical and easy to use. Besides the case studies, we have used the `gll` library to describe the syntax of ANSI-C for the experiments of Section 13.1, CBS as part of the CBS compiler, IML as part of the IML interpreter, the lambda-calculi in [Mitchell et al., 2018] and [Van Binsbergen, 2018], as well as funcon terms and configurations files within the Haskell Funcon Framework. We found these syntax descriptions easy to develop, verify and debug. And as demonstrated in Section 13.1, FUN-GLL shows acceptable running times on grammars generated from combinator expressions. The benefits of developing syntax abstractly, without having to apply factorisation or left-recursion removal, are certainly worth costlier generalised parsing. Moreover, if parsing

speed is essential, the descriptions can be refactored for efficiency.

With the BNF combinators of the `gll` package, it has been possible to define the syntax of Mini and Caml Light comfortably and concisely. A syntax description is suggestive of a grammar superficially, and when executed as a parser, the behaviour of the parser is in accordance with this grammar. The internal grammar generated from the combinator expressions, and given to the underlying parser procedure, can be analysed for the purposes of debugging or efficiency (by applying *grammarOf*). For debugging, we find that analysing the internal grammar is not necessary in practice, because individual symbol expressions can be tested in isolation and because analysing the superficial grammar is often sufficient. When a program is not recognised by the parser, the provided error messages are often informative enough to quickly find the mistake in the input string or in the syntax description. To improve parser efficiency, it may be helpful to analyse the internal grammar, especially when the details of a nonterminal's alternates are separated, as we have chosen to do in our case studies. For example, from the definition of *syn_command* alone (given in the introduction of Chapter 11), it is not possible to determine whether the grammar generated for *syn_command* is left-factored, recursive, nondeterministic, or ambiguous. We have chosen to separate the alternates of expressions, commands, and declarations so that every language construct can be introduced separately, grouping together the construct's informal explanation and its syntactic and semantic details. This makes it harder to determine the grammar generated for *syn_command*, without inspecting the internal grammar, as the necessary details are separated across pages.

The syntax descriptions of Mini and Caml Light have been developed with a focus on clarity rather than efficiency. Refactoring the syntax description to improve parser efficiency was not necessary, despite the abstract nature of the described grammar. The 50+ programs developed to test the specification of Mini are small and simple. The Caml Light syntax description enables parsing large input files, as demonstrated in Section 13.1. The types of the BNF combinators are flexible, using the type-class mechanism explained in Section 5.3.3, and they are as easy to use as conventional parser combinators. The semantic values of combinator expressions are strongly-typed, but the case studies hardly take advantage, as most semantic functions return a funcon term and all funcon terms are of type *Funcons* (more about this in §13.3.3).

Termination Although a generalised parser is guaranteed to terminate, grammar generation may not terminate, which is a significant concern whilst syntax descriptions are being developed. Nontermination is avoided in most cases by consistently inserting nonterminal names manually when symbol expressions are defined, as we have done in our case studies. However, as language definitions grow, are being revisited, or are worked on by several people, it is hard to guarantee that inserted nonterminal names are unique. Moreover, accidentally using a nonterminal name a second time will result in unexpected behaviour for which the cause may not be easy to determine. Termination of grammar generation cannot be guaranteed generally, as discussed in §3.2.3 by giving the *scales* example. When defining a recursive combinator that combines grammar fragments, care must be taken to ensure that the inserted nonterminal reflects the parameters.

A pure alternative to nonterminal insertion is provided by [Devriese and Piessens, 2012], in which primitive recursion constructs are defined with datatype generic programming. Impure alternatives typically involve automatically generated references to represent nonterminals [Gill, 2009, Claessen and Sands, 1999, Ljunglöf, 2002].

Ambiguity reduction Syntax descriptions rely on (versions of) combinators that perform ambiguity reduction during the semantic phase, e.g. *longest_match*, $\langle{:=}$, and $\langle{**}\rangle\rangle$. Combinators such as $\langle{:=}$ and $\langle{**}\rangle\rangle$ do not compromise the clarity of combinator expressions as they are variations of core combinators which behave identically outside the semantic phase. However, a combinator such as *longest_match* somewhat obfuscates the combinator expressions in which it appears, as it has no effect on the described syntax, i.e. the internal grammar. Ideally, ambiguity reduction strategies are given separately, as in CBS and SDF3/Spoofax [Kats and Visser, 2010]. This is in conflict with the combinator approach, however.

The current ambiguity reductions strategies are low-level, defined directly on EPN sets, and may not comfortably deal with certain ambiguities. Further research is required to determine which high-level strategies are necessary, to discover how these strategies are realised by filtering EPN sets, and to determine how these strategies are best made available to the user.

13.3.2 Translation

An important property of a component-based approach is that the components can be used successfully without the need to be continuously conscious of the precise details of the components' definitions, as without this property, writing formal specification would not be much easier. We believe that the case studies show that the core BNF combinators and the majority of the funcons can be used in this way, especially those combinators and funcons which occur frequently. For this reason, it is the aim that reusable components have names suggestive of their behaviour and that they are accompanied by sufficient documentation.

Shallow embedding The semantic functions associated with combinator expressions translate concrete 'flat' representations of programs into funcon terms. No intermediate abstract syntax representation of the program is formed. An advantage of such a direct translation is that the resulting language definitions are more concise.

With an intermediate representation, however, it is easier to perform static analyses, for example to determine whether syntactically valid inputs are actually valid programs. Throughout Chapter 11, we have given constraints that restrict the validity of Mini programs. Static analyses could determine whether programs satisfy these constraints. Moreover, the Caml Light case study reveals that it may be beneficial to use an intermediate representation as a vehicle for statically computing information about a program as part of its translation. For example, for the expression **let rec** *local_defs* **in** *expr* it is necessary to compute the identifiers of *local_defs* that can be bound recursively in order to translate the expression to a funcon term. However, *local_defs* needs to be translated itself as well. This has been achieved by taking $([Funcons], Funcons)$ as the semantic domain of local definitions, thus 'fusing' two computations together. This makes the semantic functions *sem_let_var*, *sem_let_pat*, and *sem_letbs* harder to reuse. With an intermediate representation, we could simply have defined a separate semantic function for each computation. A similar example is given by Caml Light's function definitions. The translation of function definitions involves **curry_n_**, currying an n-ary function. To apply **curry_n_**, the translation thus needs to compute the arity of the function being defined, thus we need two interpretations of function definitions (see *sem_fun* and *sem_mcase*).

The choice of whether to use an intermediate representation in an embedded domain

specific language is known as the choice between a shallow or deep embedding of that language. Using this terminology, we have chosen for a shallow embedding of Mini and Caml Light within Haskell, where the semantic functions take the role of syntactic operators. Theoretically, shallow and deep embeddings are closely related [Gibbons and Wu, 2014], but in practice implementations differ significantly. In general, a shallow embedding is usually easier to extend, and its implementation more succinct. With a deep embedding it is easier to perform program transformations and static computations. Techniques have been developed to overcome the practical differences between shallow and deep embeddings [Svenningsson and Axelsson, 2013, Carette et al., 2007], of which the most notable is the 'finally tagless' style [Devriese and Piessens, 2012, Carette et al., 2007].

Multiple interpretations For Caml Light, we have given different interpretations to string literals depending on whether they appear in expressions or patterns. In an expression, a string is interpreted as a **list** of **variables** holding **characters**, whereas in a pattern a string is simply a **list** of **characters**. The first interpretation (*sem_string*) is applied as part of the syntax description of constants (*syn_const*). The syntax description of constants is reused in the syntax description of patterns (*syn_pat*). To give the alternative interpretation (*string_*) to string literals in patterns, we have introduced *syn_pat_const*:

```
syn_pat_const :: SymbExpr Token Funcons
syn_pat_const = "constant-pattern" ⟨:= sem_pat_string ⟨$$⟩ string_lit
                                   ⟨||⟩ match_constant ⟨$$⟩ syn_const
match_constant p = pattern_ [closure_ [match_ [structural_assigned_ [given_], p]]]
sem_pat_string s  = match_constant (string_ s)   -- no need to initialise variables
```

A string literal is recognised by both alternatives of *syn_pat_const*, each giving an alternative interpretation. We (ab)use ambiguity reduction to indicate that the first alternative of *syn_pat_const* is preferred, and thus that the semantic function *sem_pat_string* gives the correct interpretation. This is not ideal, as we have introduced unnecessary ambiguity to the syntax description with the purpose of simplifying the funcon translation.

No interpretations Caml Light type-expressions do not influence the behaviour of programs in which they appear. However, combinator expressions must produce a semantic value. In the syntax description of type-expressions (*syn_tyexpr*), we have chosen the value

() (of type ()) which gives no information. The fact that () appears as a semantic value in most of the alternatives of *syn_tyexpr* distracts from the relevant syntactic details.

Efficiency In our experience it has not been necessary to refactor syntactic descriptions for the purposes of efficiency in order to execute test programs comfortably. The efficiency of funcon term interpretation is more precarious and, in the case of Caml Light, modifications to the funcon translation have had significant impact on the runtimes with which test programs are executed. In general, it may thus be necessary to refactor funcon translations to improve efficiency of interpretation, which is undesirable, especially when it compromises clarity. Some of these issues have been discussed in Section 13.2.

13.3.3 Funcon Implementations

The definitions of semantic functions involve applying the smart constructors exported by `Funcons.EDSL` of the `funcons-tools` package, applying smart constructors generated for language specific funcons (e.g. **mini_show_**, **caml_light_show_**, and **curry_n_**), applying other semantic functions, as well as applying auxiliary functions (e.g. *given1* and *vector_length*).

Auxiliary functions like *given1*, *vector_length*, *and_then*, and *else_cases* abbreviate common patterns and increase conciseness through code reuse. Semantic functions themselves can be reused, which makes it easy to define additional constructs as syntactic sugar (see Mini's **for**-loops or Caml Light's value definitions as examples). The majority of smart constructors is generated from CBS files, including the language specific funcons. These CBS files must be studied in order to understand the semantics of Mini and Caml Light in detail. As language definitions, Chapter 11 and 12 are therefore not self-contained. We do think that most funcon names are suggestive of their behaviour or usage, and that is therefore possible to get a good intuition about the semantics of Mini and Caml Light, without studying the funcons themselves.

Funcon term correctness The smart constructors are all[2] of type $[Funcons] \to Funcons$ and provide no information that can be exploited by a Haskell compiler to determine whether their applications are meaningful. This is a significant weakness; there are no static guar-

[2]Except those for some built-in nullary funcons such as **values**.

antees about the correctness of funcon translations. Basic arity-checking and type-checking would be able to prevent many common errors. For example, if a semantic function involves the application of several smart constructors, it is easy to misplace a closing bracket and provide a term as an argument to the wrong funcon. When writing a translation, it is up to the developer to keep track of the kinds of funcon terms produced by semantic functions. For example, all forms of Mini commands should yield **null** as a result, and Caml Light patterns should evaluate to **patterns**. These are properties that a compiler should be able to check. A consistent naming policy can be introduced as a remedy.

At the time of writing, the funcon term interpreters of the Haskell Funcon Framework do not always provide error messages that make it easy to diagnose errors in funcon terms. It has been necessary to execute programs in a step-by-step fashion (using the `--max-restart` option) to find errors whilst developing the case studies of this thesis, and whilst testing CBS' example languages. This process can be tedious and time-consuming. Although the error messages of funcon term interpreters can be improved, we expect that biggest gain is made by generating smart constructors with more informative types.

Appendix A

Generalised Parsing

A.1 Equivalence of BPTs and Big-Step Derivations

Theorem A.1.1.

$$\langle \alpha, l, r \rangle \Rightarrow_\gamma I \iff b_0 \in \mathit{bpts}(\gamma), \mathit{lb}(b_0) = \langle \alpha, l, r \rangle, \mathit{frontier}_\gamma(b_0) = I$$

Proof. Given a proof of $\langle \alpha, l, r \rangle \Rightarrow_\gamma I$, we use structural induction to show that there is a $b_0 \in \mathit{bpts}(\gamma)$ with $\mathit{lb} = \langle \alpha, l, r \rangle$ and $\mathit{frontier}_\gamma(b_0) = I_{l,r}$.

If the proof is by application of TERM, then $\alpha \in T(\gamma)$, $r = l + 1$ and $I = \alpha$. Thus $\langle \langle \alpha, l, r \rangle, [] \rangle$ is a valid BPT with frontier I.

If the proof is by application of EPS, then $\alpha = \epsilon = I$ and $l = r$. Thus $\langle \langle \alpha, l, r \rangle, [] \rangle$ is a valid BPT with frontier I.

If the proof is by application of NTERM, then $\alpha ::= \beta \in \mathit{prods}(\gamma)$, and there is a proof of $\langle \beta, l, r \rangle \Rightarrow_\gamma I$. The induction hypothesis gives a BPT b_1 with $\mathit{lb}(b_1) = \langle \beta, l, r \rangle$ and $\mathit{frontier}_\gamma(b_1) = I$. Thus $\langle \langle \alpha, l, r \rangle, b_1 \rangle$ is a valid BPT with frontier I.

If the proof is by application of SEQ, then $\alpha = s_1 \dots s_{n-1} s_n$ (with $n > 1$), there is a proof of $\langle s_1 \dots s_{n-1}, l, k \rangle \Rightarrow_\gamma u$ (for some $l \leqslant k \leqslant r$), and there is a proof of $\langle s_n, k, r \rangle \Rightarrow_\gamma v$ with $I = uv$. The induction hypothesis gives a BPT b_1 with $\mathit{lb}(b_1) = \langle s_1 \dots s_{n-1}, l, k \rangle$ and $\mathit{frontier}_\gamma(b_1) = u$, and a BPT b_2 with $\mathit{lb}(b_2) = \langle s_n, k, r \rangle$ and $\mathit{frontier}_\gamma(b_2) = v$. Thus

$\langle\langle\alpha, l, r\rangle, b_1 b_2\rangle$ is a valid BPT with frontier I.

Given a BPT b_0 with $lb(b_0) = \langle\alpha, l, r\rangle$ and $frontier_\gamma(b_0) = I$, we prove by structural induction that $\langle\alpha, l, r\rangle \Rightarrow_\gamma I$.

If b_0 has no children ($n = 0$) then $I = \alpha$ (definition of $frontier_\gamma$). If $\alpha \in T(\gamma)$ then $r = l + 1$ and we can prove $\langle\alpha, l, r\rangle \Rightarrow_\gamma I$ by applying TERM. If $\alpha = \epsilon$ then $l = r$ and we can prove $\langle\alpha, l, r\rangle \Rightarrow_\gamma I$ by applying EPS.

If b_0 has one child b_1, then $lb(b_1) = \langle\beta, l, r\rangle$, for some $\alpha ::= \beta \in prods(\gamma)$, and $frontier_\gamma(b_1) = I$. The induction hypothesis gives a proof for $\langle\beta, l, r\rangle \Rightarrow_\gamma I$ which we use in an application of NTERM to prove $\langle\alpha, l, r\rangle \Rightarrow_\gamma I$.

Finally, if b_0 has two children b_1 and b_2, then $\alpha = s_1 \ldots s_{n-1} s_n$ (with $n > 1$), $lb(b_1) = \langle s_1 \ldots s_{n-1}, l, k\rangle$ (for some k), $frontier_\gamma(b_1) = u$, $lb(b_2) = \langle s_n, k, r\rangle$, and $frontier_\gamma(b_2) = v$ with $I = uv$. The induction hypothesis gives proofs for $\langle s_1 \ldots s_{n-1}, l, k\rangle \Rightarrow_\gamma u$ and $\langle s_n, k, r\rangle \Rightarrow_\gamma v$ which are used in an application of SEQ to prove $\langle\alpha, l, r\rangle \Rightarrow_\gamma I$.

□

A.2 Proof of Completeness CDS

Theorem A.2.1. For any grammar γ and string I it holds that $\langle\mathcal{U}(\gamma, I), \mathcal{E}\rangle = cds(\gamma, I)$

Proof. The obligation is to show $\mathcal{U}^C = \mathcal{U}(\gamma, I)$ for any grammar γ and string I, where $\langle\mathcal{U}^C, \mathcal{E}\rangle = cds(\gamma, I)$. Every element in $\mathcal{U}^C$ was in $\mathcal{R}$ in some call of $loop_{\gamma,I}$ and vice versa. By construction, *cds* is conservative and adds only an item d to $\mathcal{R}$ if one the rules $R(1) - R(4)$ of Definition 2.2.2 justifies it, proving the minimality of $\mathcal{U}^C$. More complicated is the proof that no items have been omitted. The initialisation of $\mathcal{R}$ in the first call to $loop_{\gamma,I}$ shows $R(1)$ holds on $\mathcal{U}^C$ (see Definition 2.2.2). For any $d \in \mathcal{U}^C$ there is a call $loop_{\gamma,I}(\mathcal{R}, \mathcal{U}, \mathcal{E})$ with $d \in \mathcal{R}$ and $d \notin \mathcal{U}$ and thus a call $process_{\gamma,I}(d, \mathcal{U} \cup \{d\})$ (because items are only added to $\mathcal{U}$ as part of their processing). $R(2)$ holds for $\mathcal{U}^C$, as follows from the call and definition of $match_{\gamma,I}$ for each descriptor $\langle X ::= \alpha \cdot t\beta, l, k\rangle \in \mathcal{U}^C$ (with t terminal). $R(3)$ holds for $\mathcal{U}^C$, as follows from the call and definition of $descend_{\gamma,I}$ for each descriptor $\langle X ::= \alpha \cdot Y\beta, l, k\rangle \in \mathcal{U}^C$ (with Y nonterminal). To prove that $R(4)$ holds for $\mathcal{U}^C$, we assume arbitrary $d_1 = \langle X ::= \alpha \cdot Y\beta, l, k\rangle \in \mathcal{U}^C$ and $d_2 = \langle Y ::= \delta\cdot, k, r\rangle \in \mathcal{U}^C$ and show $d_3 = \langle X ::= \alpha Y \cdot \beta, l, r\rangle \in \mathcal{U}^C$ is implied. There are four cases to consider: there is a call to $loop_{\gamma,I}$ with $d_1 = select(\mathcal{R})$

and $d_2 \in \mathcal{U}$, a call with $d_1 \in \mathcal{U}$ and $d_2 = \mathit{select}(\mathcal{R})$, a call with $d_1 = \mathit{select}(\mathcal{R})$ and $d_2 \notin \mathcal{U}$, and finally a call with $d_1 \notin \mathcal{U}$ and $d_2 = \mathit{select}(\mathcal{R})$. In the first two cases, $d_3 \in \mathcal{U}^C$ follows from the call and definition of $\mathit{skip}_{\gamma,I}$ and $\mathit{ascend}_{\gamma,I}$ respectively. In the third case, even though $d_2 \notin \mathcal{U}$ in the 'current' call to $\mathit{loop}_{\gamma,I}$ we know, because $d_2 \in \mathcal{U}^C$, that there must be a future call to $\mathit{loop}_{\gamma,I}$ which corresponds to the second of our cases (because by that time $d_1 \in \mathcal{U}$). In the fourth case, even though $d_1 \notin \mathcal{U}$ in the 'current' call to $\mathit{loop}_{\gamma,I}$ we know, because $d_1 \in \mathcal{U}^C$, that there must be a future call to $\mathit{loop}_{\gamma,I}$ which corresponds to the first of our cases (because by that time $d_2 \in \mathcal{U}$). □

A.3 Proving CDS Computes a Complete Set of EPNs

Theorem A.3.1. Given a grammar γ and a string I, $\mathcal{E}^C = \Delta(\mathcal{U}(\gamma, i))$ when $\langle \mathcal{U}^C, \mathcal{E}^C \rangle = \mathit{cds}(\gamma, I)$.

Proof. According to Theorem A.2.1, $\mathcal{U}^C = \mathcal{U}(\gamma, I)$, thus every element in $\mathcal{U}(\gamma, I)$ is processed in some call of $\mathit{loop}_{\gamma,I}$ by the call $\mathit{cds}(\gamma, I)$.

A packed node of the form $\langle Y ::= \cdot, k, r, r \rangle$ is in $\mathcal{E}^C$ only if, and whenever, an item of the form $\langle Y ::= \cdot, k, r \rangle$ is processed. Thus there is no spurious EPN of that form in $\mathcal{E}^C$ and $P(3)$ is satisfied (see Definition 2.2.3).

A packed node of the form $\langle X ::= \alpha t \cdot \beta, l, k, k+1 \rangle$ (with t terminal) is added to $\mathcal{E}^C$ only if, and whenever, an item of the form $\langle X ::= \alpha \cdot t\beta, l, k \rangle$ is processed and $I_k = t$. Thus there is no spurious EPN of that form in $\mathcal{E}^C$ and $P(1)$ is satisfied.

A packed node of the form $\langle X ::= \alpha Y \cdot \beta, l, k, r \rangle$ (with Y nonterminal) is added to $\mathcal{E}^C$ only when an item of the form $\langle X ::= \alpha \cdot Y, l, k \rangle$ is processed whilst there is an item $\langle Y ::= \delta \cdot, k, r \rangle$ in $\mathcal{U}$, or when an item of the form $\langle Y ::= \delta \cdot, k, r \rangle$ is processed whilst an item of the form $\langle X ::= \alpha \cdot Y\beta, l, k \rangle$ is in $\mathcal{U}$. Thus there is no spurious EPN of the form $\langle X ::= \alpha Y \cdot \beta, l, k, r \rangle$ in $\mathcal{E}^C$.

To prove that $P(2)$ holds, we show that any $d_1 = \langle X ::= \alpha \cdot Y\beta, l, k \rangle \in \mathcal{U}^C$ and $d_2 = \langle Y ::= \delta \cdot, k, r \rangle \in \mathcal{U}^C$ imply that $e_1 = \langle X ::= \alpha Y \cdot \beta, l, k, r \rangle \in \mathcal{E}^C$. There are four cases to consider: there is a call to $\mathit{loop}_{\gamma,I}$ with $d_1 = \mathit{select}(\mathcal{R})$ and $d_2 \in \mathcal{U}$, a call with $d_1 \in \mathcal{U}$ and $d_2 = \mathit{select}(\mathcal{R})$, a call with $d_1 = \mathit{select}(\mathcal{R})$ and $d_2 \notin \mathcal{U}$, and finally a call with $d_1 \notin \mathcal{U}$ and $d_2 = \mathit{select}(\mathcal{R})$. In the first two cases, $e_3 \in \mathcal{E}^C$ follows from the call and definition of

skip$_{\gamma,I}$ and *ascend*$_{\gamma,I}$ respectively. In the third case, even though $d_2 \notin \mathcal{U}$ in the 'current' call to *loop*$_{\gamma,I}$ we know, because $d_2 \in \mathcal{U}^C$, that there must be a future call to *loop*$_{\gamma,I}$ which corresponds to the second of our cases (because by that time $d_1 \in \mathcal{U}$). In the fourth case, even though $d_1 \notin \mathcal{U}$ in the 'current' call to *loop*$_{\gamma,I}$ we know, because $d_1 \in \mathcal{U}^C$, that there must be a future call to *loop*$_{\gamma,I}$ which corresponds to the first of our cases (because by that time $d_2 \in \mathcal{U}$). □

A.4 Building BSPPFs from Extended Packed Nodes

The Binarised Shared Packed Parse Forest (BSPPF) is a specialised data structure using sharing to represent all possible derivations of some string in $O(n^3)$ space. State-of-the-art complete parsers can construct a BSPPF in worst-case $O(n^3)$ time [Tomita, 1985, Scott and Johnstone, 2010b, Scott and Johnstone, 2013].

Nodes in BSPPFs are labelled with a symbol, a left extent and a right extent (symbol nodes); a grammar slot, a left extent and a right extent (intermediate node); or a grammar slot and a pivot (packed node). The definitions of BSPPFs in [Scott and Johnstone, 2010b, Scott and Johnstone, 2013] also mention epsilon nodes, but we do not use them here. We denote symbol nodes, intermediate nodes, and packed nodes as (s, l, r), $(X ::= \alpha \cdot \beta, l, r)$, and $(X ::= \alpha \cdot \beta, k)$ respectively, for arbitrary choices of $s \in S$, $l, k, r \in \mathbb{N}$, and $X ::= \alpha \cdot \beta$ with $X ::= \alpha\beta \in S \times S^*$.

Intermediate nodes, and symbol nodes labelled with a nonterminal, have one or more packed node children. The intuition is that each packed node represents particular choices leading to the successful recognition of a string. Thus, multiple packed nodes under the same node implies that multiple choices lead to success.

A packed node has either zero, one or two children. If it has no children, the packed node is labelled with a slot of the form $X ::= \cdot$ (the only slot for the production $X ::= \epsilon$). If it has one child, the packed node is labelled with a slot of the form $X ::= s \cdot \beta$, with $s \in S$, and the child is a symbol node labelled with s. If a packed node has two children, it is labelled with a slot of the form $X ::= \alpha s \cdot \beta$ and some pivot k ($\alpha \neq \epsilon$). Its first child is an intermediate node with label $X ::= \alpha \cdot s\beta$, left extent l (from the grandparent), and right extent k. The second child is a symbol node labelled with s, left extent k and right extent

r (from the grandparent). A BSPPF has one or more symbol nodes as roots.

Figure A.1 shows an BSPPF representing both parse trees of Figure 2.2. The packed nodes labelled e_7 and e_6 represent the two possible choices for k when proving $\langle AA, 0, 1\rangle \Rightarrow_\gamma a$ by applying rule IM.

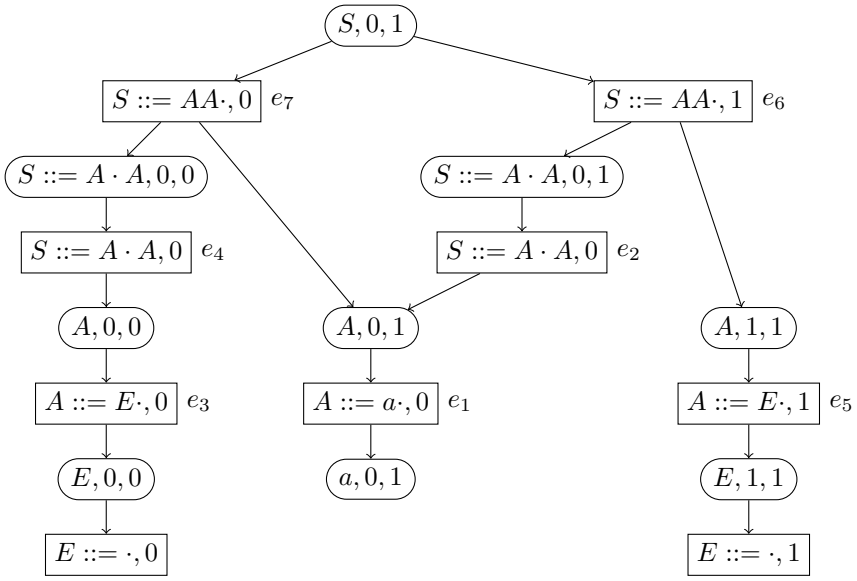


Figure A.1: BSPPF embedding both parse trees of Figure 2.2.

A.4.1 The Construction

Given a set of extended packed nodes Δ, a BSPPF is constructed by performing the following actions for each extended packed node in $(X ::= \alpha s\cdot\beta, l, k, r) \in \Delta$ (in any order). (Extended packed nodes of the form $(X ::= \cdot\beta, l, k, r)$ can be ignored.)

1. Create packed node $p = (X ::= \alpha s \cdot \beta, k)$, and:

   - If $\beta = \epsilon$, make p a child of the symbol node (X, l, r) (which may need to be created if it does not exist)
   - If $\beta \neq \epsilon$, make p a child of the intermediate node $(X ::= \alpha s \cdot \beta, l, r)$ (which may need to be created if it does not exist)

$$
\begin{aligned}
e_1 &= (A ::= a\cdot, 0, 0, 1)\\
e_2 &= (S ::= A \cdot A, 0, 0, 1)\\
e_3 &= (A ::= E\cdot, 0, 0, 0)\\
e_4 &= (S ::= A \cdot A, 0, 0, 0)\\
e_5 &= (A ::= E\cdot, 1, 1, 1)\\
e_6 &= (S ::= AA\cdot, 0, 1, 1)\\
e_7 &= (S ::= AA\cdot, 0, 0, 1)
\end{aligned}
$$

Figure A.2: Extended packed nodes for constructing the BSPPF from Figure A.1.

2. If $\alpha = \epsilon$, then the symbol node $n = (s, l, r)$ is the only child of p (n may need to be created if it does not exist)

3. If $\alpha \neq \epsilon$, then the intermediate node $i = (X ::= \alpha \cdot s\beta, l, k)$ is the first child of p and symbol node $n = (s, k, r)$ is the second child of p (i and n may need to be created if they do not exist)

4. If a symbol node n was created with nonterminal label X, left extent l, right extent $r = l$ and if there is a production $X ::= \epsilon$, then create a packed node $q = (X ::= \cdot, l)$ and add it to the children of n unless n already has such a child

The BSPPF of Figure A.1 is constructed from the set of extended packed nodes given in Figure A.2. A (regular) packed node in Figure A.1 has been labelled when it corresponds to an extended packed node. The extended packed node is the regular packed node extended with the left and right extent of its parent.

A.4.2 Avoiding Spurious Derivations

The following recursive procedure produces a trace of extended packed nodes. Given a set Δ of extended packed nodes and an element of Δ of the form $(X ::= \alpha \cdot \beta, l, k, r)$:

1. Do nothing if $\alpha = \epsilon$

2. If $\alpha = \alpha_1 s$, make a recursive call with Δ for all extended packed nodes $(X ::= \alpha_1 \cdot s\beta, l, k_1, k) \in \Delta$ and for all extended packed nodes $(s ::= \beta_0, k, k_2, r) \in \Delta$

Consider the traces of extended packed nodes produced by applying this procedure to $\Delta(\mathcal{U}(\gamma, I))$ and all extended packed nodes of the form $(Z ::= \alpha\cdot, 0, k, |I|) \in \Delta(\mathcal{U}(\gamma, I))$, for some grammar γ with start symbol Z and input string I. If the procedure of Section A.4.1 is applied to all extended packed nodes in these traces, then the resulting BSPPF embeds all derivations of I according to γ but embeds no spurious derivations.

Bibliography

[Afroozeh et al., 2013] Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., and Vinju, J. (2013). Safe specification of operator precedence rules. In *Software Language Engineering: 6th International Conference*, SLE 2013, pages 137–156. Springer International Publishing.

[Astesiano, 1991] Astesiano, E. (1991). Inductive and operational semantics. In Neuhold, E. and Paul, M., editors, *IFIP State-of-the-Art Reports, Formal Descriptions of Programming Concepts*, pages 51–136. Springer.

[Aycock and Horspool, 2002] Aycock, J. and Horspool, R. N. (2002). Practical earley parsing. *The Computer Journal*, 45(6):620–630.

[Baars and Swierstra, 2004] Baars, A. I. and Swierstra, S. D. (2004). Type-safe, self inspecting code. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell 2004, pages 69–79. ACM.

[Bach Poulsen, 2016] Bach Poulsen, C. (2016). *Extensible Transition System Semantics*. PhD thesis, Swansea University.

[Bach Poulsen and Mosses, 2014a] Bach Poulsen, C. and Mosses, P. D. (2014a). Deriving pretty-big-step semantics from small-step semantics. In *Programming Languages and Systems: 23rd European Symposium on Programming*, ESOP 2014, pages 270–289. Springer Berlin Heidelberg.

[Bach Poulsen and Mosses, 2014b] Bach Poulsen, C. and Mosses, P. D. (2014b). Generating specialized interpreters for modular structural operational semantics. In *23rd Inter-*

national Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2013, pages 220–236. Springer.

[Benzaken et al., 2003] Benzaken, V., Castagna, G., and Frisch, A. (2003). ℂDuce: An xml-centric general-purpose language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP 2003. ACM.

[Berger and Tratt, 2010] Berger, M. and Tratt, L. (2010). Program logics for homogeneous meta-programming. In Clarke, E. M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 64–81. Springer Berlin Heidelberg.

[Berger et al., 2017] Berger, M., Tratt, L., and Urban, C. (2017). Modelling homogeneous generative meta-programming. In *Proceedings of the 31st European Conference on Object-Oriented Programming*.

[van Binsbergen, 2018] van Binsbergen, L. T. (2018). Funcons for HGMP - the fundamental constructs of homogeneous generative meta-programming (short paper). In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts & Experience*, GPCE 2018.

[van Binsbergen et al., 2019] van Binsbergen, L. T., Mosses, P. D., and Sculthorpe, N. (2019). Executable component-based semantics. *Journal of Logical and Algebraic Methods in Programming*, 103:184–212.

[van Binsbergen et al., 2018] van Binsbergen, L. T., Scott, E., and Johnstone, A. (2018). GLL parsing with flexible combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018.

[van Binsbergen et al., 2016] van Binsbergen, L. T., Sculthorpe, N., and Mosses, P. D. (2016). Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 8–11. ACM.

[van den Brand et al., 2003] van den Brand, M. G., Klusener, A., Moonen, L., and Vinju, J. (2003). Generalized parsing and term rewriting: Semantics driven disambiguation. *Electronic Notes in Theoretical Computer Science*, 82(3):575 – 591.

[Carette et al., 2007] Carette, J., Kiselyov, O., and Shan, C.-C. (2007). Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS 2007, pages 222–238. Springer-Verlag.

[Churchill and Mosses, 2013] Churchill, M. and Mosses, P. D. (2013). Modular bisimulation theory for computations and values. In *Foundations of Software Science and Computation Structures*, pages 97–112. Springer Berlin Heidelberg.

[Churchill et al., 2015] Churchill, M., Mosses, P. D., Sculthorpe, N., and Torrini, P. (2015). Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development XII*, TAOSD 2015, pages 132–179.

[Claessen and Sands, 1999] Claessen, K. and Sands, D. (1999). Observable sharing for functional circuit description. In *In Asian Computing Science Conference*, pages 62–73. Springer Verlag.

[Danvy and Nielsen, 2004] Danvy, O. and Nielsen, L. (2004). Refocusing in reduction semantics. *BRICS Report Series*, 11(26).

[Day and Hutton, 2012] Day, L. E. and Hutton, G. (2012). Towards modular compilers for effects. In *12th International Symposium on Trends in Functional Programming*, volume 7193 of *Lecture Notes in Computer Science*, pages 49–64. Springer.

[Devriese and Piessens, 2011] Devriese, D. and Piessens, F. (2011). Explicitly recursive grammar combinators. In *Practical Aspects of Declarative Languages*, PADL 2011, pages 84–98.

[Devriese and Piessens, 2012] Devriese, D. and Piessens, F. (2012). Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22(6):757–796.

[Duregård and Jansson, 2011] Duregård, J. and Jansson, P. (2011). Embedded parser generators. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell 2011. ACM.

[Earley, 1970] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

[Felleisen et al., 2009] Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition.

[Frost et al., 2008] Frost, R. A., Hafiz, R., and Callaghan, P. (2008). Parser combinators for ambiguous left-recursive grammars. In *Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin Heidelberg.

[Gibbons and Wu, 2014] Gibbons, J. and Wu, N. (2014). Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2014, pages 339–347.

[Gill, 2009] Gill, A. (2009). Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell 2009, pages 117–128. ACM.

[Gordon, 1995] Gordon, A. D. (1995). A tutorial on co-induction and functional programming. In Hammond, K., Turner, D. N., and Sansom, P. M., editors, *Functional Programming, Glasgow 1994*, pages 78–95. Springer London.

[Groote, 1993] Groote, J. F. (1993). Transition system specifications with negative premises. *Theoretical Computer Science*, 118(2):263 – 299.

[Grune, 2010] Grune, D. (2010). *Parsing Techniques: A Practical Guide*. Springer Publishing Company, Incorporated, 2nd edition.

[Hall et al., 1994] Hall, C., Hammond, K., Peyton Jones, S., and Wadler, P. (1994). Type classes in haskell. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256. Springer Verlag.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

[Hudak et al., 2007] Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. (2007). A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, New York, NY, USA. ACM.

[Izmaylova et al., 2016] Izmaylova, A., Afroozeh, A., and van der Storm, T. (2016). Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, pages 1–12. ACM.

[Johnson, 1995] Johnson, M. (1995). Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417.

[Johnstone and Scott, 2011] Johnstone, A. and Scott, E. (2011). Modelling gll parser implementations. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE 2010, pages 42–61. Springer-Verlag.

[Jones, 1995] Jones, M. P. (1995). Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136. Springer-Verlag.

[Kats and Visser, 2010] Kats, L. C. L. and Visser, E. (2010). The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2010, pages 444–463. ACM.

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall.

[Lazar et al., 2012] Lazar, D., Arusoaie, A., Şerbănuţă, T. F., Ellison, C., Mereuta, R., Lucanu, D., and Roşu, G. (2012). *Executing Formal Semantics with the $\mathbb{K}$ Tool*, volume 7436 of *Lecture Notes in Computer Science*, pages 267–271. Springer Berlin Heidelberg.

[Leijen and Meijer, 2001] Leijen, D. and Meijer, E. (2001). Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht.

[Leroy, 1997] Leroy, X. (1997). Caml Light manual. `http://caml.inria.fr/pub/docs/manual-caml-light`.

[Liang et al., 1995] Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *22nd Symposium on Principles of Programming Languages*, pages 333–343. ACM.

[Ljunglöf, 2002] Ljunglöf, P. (2002). *Pure Functional Parsing*. PhD thesis, Chalmers University of Technology and Göteborg University.

[Mcbride and Paterson, 2008] Mcbride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming.*, 18(1):1–13.

[Mitchell et al., 2018] Mitchell, D., van Binsbergen, L. T., Loring, B., and Kinder, J. (2018). Checking cryptographic API usage with composable annotations (short paper). In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2018. ACM.

[Mosses, 1990] Mosses, P. (1990). Denotational semantics. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*, pages 577–631. MIT Press.

[Mosses, 1999] Mosses, P. D. (1999). A modular SOS for ML concurrency primitives. *BRICS Report Series*, (57).

[Mosses, 2004] Mosses, P. D. (2004). Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228.

[Mosses and New, 2009] Mosses, P. D. and New, M. J. (2009). Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4).

[Norvig, 1991] Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98.

[Okasaki and Gill, 1998] Okasaki, C. and Gill, A. (1998). Fast mergeable integer maps. In *In Workshop on ML*, pages 77–86.

[Pettersson, 1999] Pettersson, M. (1999). *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer.

[Peyton Jones, 1987] Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall.

[Pingali and Bilardi, 2015] Pingali, K. and Bilardi, G. (2015). A graphical model for context-free grammar parsing. In Franke, B., editor, *Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 3–27. Springer Berlin Heidelberg.

[Plotkin, 2004a] Plotkin, G. D. (2004a). The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3 – 15.

[Plotkin, 2004b] Plotkin, G. D. (2004b). A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139. Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981.

[Rhiger, 2009] Rhiger, M. (2009). Type-safe pattern combinators. *Journal of Functional Programming*, 19(2):145–156.

[Ridge, 2014] Ridge, T. (2014). Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle. In *International Conference on Software Language Engineering, SLE 2014*, volume 8706 of *Lecture Notes in Computer Science*, pages 261–281. Springer International Publishing.

[Roşu and Şerbănuţă, 2014] Roşu, G. and Şerbănuţă, T. F. (2014). K overview and simple case study. *Electronic Notes in Theoretical Computer Science*, 304:3 – 56.

[Scott and Johnstone, 2010a] Scott, E. and Johnstone, A. (2010a). GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177 – 189. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[Scott and Johnstone, 2010b] Scott, E. and Johnstone, A. (2010b). Recognition is not parsing - SPPF-style parsing from cubic recognisers. *Science of Computer Programming*, 75(1-2):55–70.

[Scott and Johnstone, 2013] Scott, E. and Johnstone, A. (2013). GLL parse-tree generation. *Science of Computer Programming*, 78(10):1828 – 1844.

[Scott and Johnstone, 2016] Scott, E. and Johnstone, A. (2016). Structuring the GLL parsing algorithm for performance. *Science of Computer Programming*, 125:1 – 22.

[Scott et al., 2019] Scott, E., Johnstone, A., and van Binsbergen, L. T. (2019). Derivation representation using binary subtree sets. *Science of Computer Programming*, 175:63–84.

[Scott et al., 2007] Scott, E., Johnstone, A., and Economopoulos, R. (2007). BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461.

[Sculthorpe et al., 2016] Sculthorpe, N., Torrini, P., and Mosses, P. D. (2016). A modular structural operational semantics for delimited continuations. In *Post-Proceedings of the 2015 Workshop on Continuations*, volume 212 of *Electronic Proceedings in Theoretical Computer Science*, pages 63–80. Open Publishing Association.

[Sheard and Peyton Jones, 2002] Sheard, T. and Peyton Jones, S. (2002). Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75.

[Svenningsson and Axelsson, 2013] Svenningsson, J. and Axelsson, E. (2013). Combining deep and shallow embedding for EDSL. In *Internal Symposium on Trends in Functional Programming, TFP 2012*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36.

[Swierstra, 2009] Swierstra, S. D. (2009). Combinator parsing: A short tutorial. In Bove, A., Barbosa, L. S., Pardo, A., and Pinto, J. S., editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 252–300. Springer Berlin Heidelberg.

[Swierstra and Duponcheel, 1996] Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In Launchbury, J., Meijer, E., and Sheard, T., editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer Berlin Heidelberg.

[Swierstra, 2008] Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programming.*, 18(4):423–436.

[Taha and Sheard, 2000] Taha, W. and Sheard, T. (2000). MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211 – 242. PEPM 1997.

[The $\mathbb{K}$ Framework, 2018] The $\mathbb{K}$ Framework (2018). The $\mathbb{K}$ tools. GitHub: https://github.com/kframework/k5. [Online, accessed 7th September 2018].

[Tomita, 1985] Tomita, M. (1985). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

[Tratt, 2005] Tratt, L. (2005). Compile-time meta-programming in a dynamically typed OO language. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS 2005, pages 49–63. ACM.

[Vergu et al., 2015] Vergu, V. A., Neron, P., and Visser, E. (2015). DynSem: A DSL for dynamic semantics specification. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015*, volume 36 of *Leibniz International Proceedings in Informatics*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Visser, 2001] Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. In *12th International Conference on Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer.

[Wadler, 1985] Wadler, P. (1985). How to replace failure by a list of successes. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer Berlin Heidelberg.

[Wu et al., 2014] Wu, N., Schrijvers, T., and Hinze, R. (2014). Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell 2014, pages 1–12. ACM.